

Erich GAMMA
Richard Helm

Este libro no trata de una introducción a la tecnología orientada a objetos ni al diseño orientado a objetos. Ya hay muchos libros que sirven bien a ese propósito.

Por otro lado, tampoco es éste un avanzado tratado técnico. Es un libro de patrones de diseño que describe soluciones simples y elegantes para problemas específicos del diseño de software orientado a objetos.

En este libro, Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides introducen los principios de los patrones de diseño y ofrecen un catálogo de dichos patrones. Así, este libro realiza dos importantes contribuciones. En primer lugar, muestra el papel que los patrones pueden desempeñar diseñando la arquitectura de sistemas complejos. En segundo lugar, proporciona una referencia práctica de un conjunto de excelentes patrones que el desarrollador puede aplicar para construir sus propias aplicaciones.

Una advertencia y unas palabras de ánimo: no se preocupe si no entiende del todo este libro en la primera lectura. (Recuerde que no es un libro para leer una vez y después ponerlo en una estantería)

Esperamos que acuda a él una y otra vez en busca de pistas de diseño y de inspiración.



Erich Gamma & Richard Helm & Ralph Johnson & John Vlissides

Patrones de Diseño

Elementos de software orientado a objetos reutilizable

ePub r1.0

XcUiDi 13.12.2018

Título original: *Design Patterns: Elements of Reusable Object-Oriented Software*
Erich Gamma & Richard Helm & Ralph Johnson & John Vlissides, 1995
Traducción: César Fernández Acebal

Editor digital: XcUiDi
ePub base r2.0

A Karin
E. G.

A Sylvie
R. H.

A Faith
R. J.

A Dru Attn y Matthew
Josué 24,15b
J. V.

Elogios a

Patrones de Diseño: Elementos de Software Orientado a Objetos Reutilizable

"Éste es uno de los libros mejor escritos y más maravillosamente perspicaces que he leído en mucho tiempo… Este libro establece la legitimidad de los patrones del mejor modo: no mediante razonamientos sino mediante ejemplos".

Stan Lippman,
C+ + Report

"…este nuevo libro de Gamma, Helm, Johnson y Vlissides promete tener un impacto importante y definitivo en la disciplina del diseño de software. Debido a que Patrones de Diseño *se anuncia a sí mismo como relacionado sólo con software orientado a objetos, temo que los desarrolladores de software no pertenecientes a la comunidad de objetos puedan pasarlo por alto. Sería una pena. Este libro tiene algo para todos aquellos que diseñan software. Todos los diseñadores de software usan patrones: entender mejor las abstracciones reutilizables de nuestro trabajo sólo puede hacemos mejores en él".*

Tom DeMarco,
IEEE Software

"Creo que este libro representa una contribución extremadamente valiosa y única en este campo porque capta una riqueza de experiencia de diseño orientado a objetos en una forma compacta y reutilizable. Este libro es seguramente uno al que acudiré a menudo en búsqueda de poderosas ideas de diseño orientado a objetos; después de todo, de eso es de lo que trata la reutilización, ¿no es cierto?".

Sanjiv Gossain,
Journal of Object-Oriented Programming

"Este libro largamente esperado está a la altura de todo el año en que ha estado en boga anticipadamente. La metáfora es la de un libro de patrones de un arquitecto lleno de diseños probados y utilizables. Los autores han elegido 23 patrones procedentes de décadas de experiencia en orientación a objetos. La brillantez de este libro reside en la disciplina representada por ese número. Dele una copia de Patrones de Diseño *a todo buen programador que conozca que quiera ser mejor".*

Larry O'Brien,
Software Development

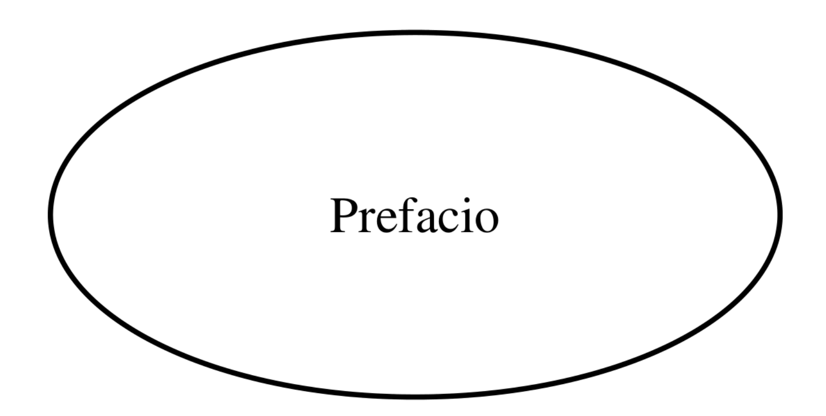
"El hecho es que los patrones tienen el potencial de cambiar para siempre el campo de la ingeniería del software, catapultándola al reino de un diseño realmente elegante. De los libros sobre este tema hasta la fecha, Patrones de Diseño *es con mucho el mejor. Es un libro para ser leído, estudiado, interiorizado y querido. Cambiará para siempre el modo en que ve el software".*

Steve Bilow,

Journal of Object Oriented Programming

“Patrones de Diseño es un libro poderoso. Después de invertir un poco de tiempo en él, la mayoría de los programadores de C++ serán capaces de comenzar a aplicar sus 'patrones' para producir mejor software. Este libro suministra un capital intelectual: herramientas concretas que nos ayudan a pensar y expresamos de forma más efectiva. Puede cambiar drásticamente nuestra forma de pensar en la programación”.

Tom Cargill,
C++ Report



Prefacio

Este libro no es una introducción a la tecnología o el diseño orientados a objetos. Ya hay muchos libros que sirven bien a ese propósito. Este libro presupone que el lector domina razonablemente al menos un lenguaje de programación orientado a objetos, y sería igualmente deseable que tuviese algo de experiencia en el diseño orientado a objetos. En definitiva, que no se abalanzará sobre el diccionario más cercano cuando hablemos de “tipos” y “polimorfismo”, o de la herencia de “interfaces” frente a la de “implementación”.

Por otro lado, tampoco es éste un avanzado tratado técnico. Es un libro de **patrones de diseño** que describe soluciones simples y elegantes a problemas específicos del diseño de software orientado a objetos. Los patrones de diseño representan soluciones que han sido desarrolladas y han ido evolucionando a lo largo del tiempo. Por tanto, no se trata de los diseños que la gente tiende a generar inicialmente, sino que reflejan todo el rediseño y la recodificación que los desarrolladores han ido haciendo a medida que luchaban por conseguir mayor reutilización y flexibilidad en su software. Los patrones de diseño expresan estas soluciones de una manera sucinta y fácilmente aplicable.

Los patrones de diseño no requieren características sofisticadas del lenguaje ni sorprendentes trucos de programación con los que dejar atónitos a jefes y amigos. Todo se puede implementar en lenguajes orientados a objetos estándar, si bien puede costar algo más de trabajo que las soluciones *ad hoc*. Pero este esfuerzo extra

siempre se ve recompensado por un aumento de la flexibilidad y la reutilización.

Una vez que haya entendido los patrones de diseño y experimentado con ellos, nunca más pensará en el diseño orientado a objetos de la misma manera que antes. Tendrá una perspectiva que puede hacer que sus propios diseños sean más flexibles, modulares, reutilizables y comprensibles, lo cual, al fin y al cabo, es la principal razón por la que está interesado en la tecnología orientada a objetos, ¿verdad?

Una advertencia y unas palabras de ánimo: no se preocupe si no entiende del todo este libro en la primera lectura. ¡Tampoco nosotros lo entendimos todo la primera vez que lo escribimos! Recuerde que no es un libro para leer una vez y ponerlo después en una estantería. Esperamos que acuda a él una y otra vez en busca de pistas de diseño y de inspiración. Este libro ha tenido un largo periodo de gestación. Ha visto cuatro países, los matrimonios de tres de sus autores y el nacimiento de dos hijos. Mucha gente ha participado en su desarrollo. Estamos especialmente agradecidos a Bruce Anderson, Kent Beck, y André Weinand por su inspiración y consejos. También a aquellos que revisaron los borradores del original: Roger Bielefeld, Grady Booch, Tom Cargill, Marshall Cline, Ralph Hyre, Brian Kemighan, Thomas Laliberty, Mark Lorenz, Arthur Riel, Doug Schmidt, Clovis Tondo, Steve Vinoski y Rebecca Wirfs-Brock. También queremos agradecer al equipo de Addison-Wesley su ayuda y paciencia: Kate Habib, Tiffany Moore, Lisa Raffaele, Pradeepa Siva y John Wait. Muchas gracias a Carl Kessler, Danny Sabbah, y Mark Wegman, de IBM Research, por su apoyo constante a este trabajo.

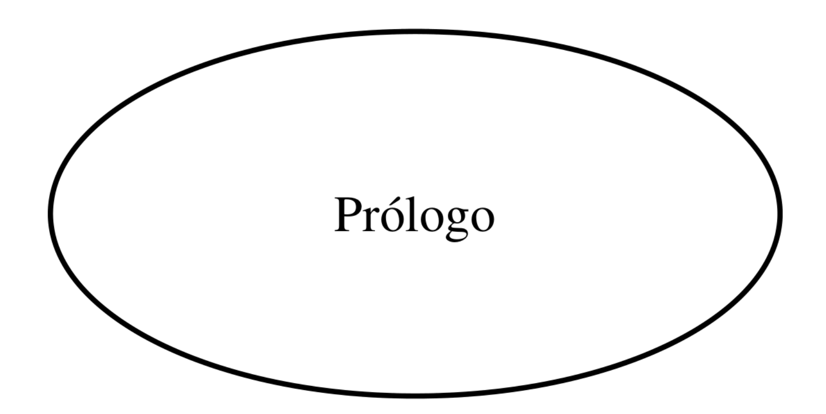
Por último, pero no menos importante, gracias a todos los que dentro y fuera de Internet comentaron versiones de los patrones, nos dieron palabras de aliento y nos dijeron que lo que estábamos haciendo valía la pena. Éstas personas son, entre otras, Jon Avotins, Steve Berczuk, Julian Berdych, Matthias Bohlen, John Brant, Allan Clarke, Paul Chisholm, Jens Coldewey, Dave Collins, Jim Coplien, Don Dwiggins, Gabriele Elia, Doug Felt, Brian Foote, Denis Fortin, Ward Harold, Hermann Hueni, Nayeem Islam, Bikramjit Kalra, Paul Keefer, Thomas Kofler, Doug Lea, Dan LaLiberte, James Long, Ann Louise Luu, Pundi Madhavan, Brian Marick, Robert Martin, Dave

McComb, Carl McConnell. Christine Mingins. Hanspeter Móssenbock, Eric Newton, Marianne Ozkan. Roxsan Payette, Larry Podmolik, George Radin. Sita Ramakrishnan, Russ Ramirez, Alexander Ran, Dirk Riehle. Bryan Rosenburg, Aamod Sane, Duri Schmidt, Robert Seidl, Xin Shu y Bill Walker.

No consideramos a esta colección de patrones de diseño como algo completo y estático: se trata más bien de una recopilación de nuestras ideas actuales sobre diseño. Los comentarios al respecto serán bienvenidos, ya sean críticas de nuestros ejemplos, referencias y usos conocidos que hayamos pasado por alto u otros patrones que deberíamos haber incluido. Puede escribirnos un correo electrónico a *design-patterns-source@cs.uiuc.edu*. También puede obtener una copia del código de las secciones de ejemplo enviándonos el mensaje "send design pattern source" a *design-patterns-source@cs.uiuc.edu*. Y existe una página Web en http://st-www.cs.uiuc.edu/users/patterns/DPBook/DPBook.html donde podrá encontrar información de última hora y actualizaciones.

Mountain View, California E.G.
Montreal, Quebec R.H.
Urbana, Illinous R.J.
Hawthorne, Nueva York J.V.

Agosto de 1994



Prólogo

Todas las arquitecturas orientadas a objetos que están bien estructuradas están repletas de patrones. De hecho, uno de los métodos que utilizo para medir la calidad de un sistema orientado a objetos es examinar si los desarrolladores le han prestado la debida atención a las colaboraciones entre sus objetos. Centrarse en tales mecanismos durante el desarrollo de un sistema puede dar lugar a una arquitectura más pequeña, simple y mucho más comprensible que si se hubieran obviado los patrones.

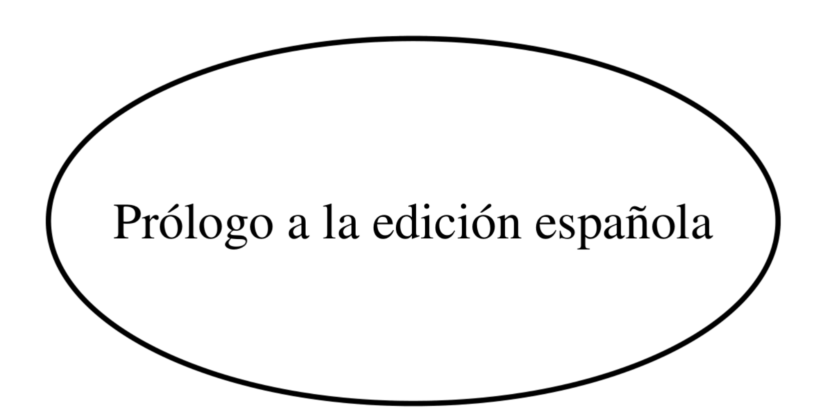
La importancia de los patrones en la construcción de sistemas complejos ha sido reconocida desde hace tiempo en otras disciplinas. En concreto, Christopher Alexander y sus colegas fueron quizá los primeros en proponer la idea de emplear un lenguaje de patrones para diseñar la arquitectura de edificios y ciudades. Sus ideas y las contribuciones de otros están ahora arraigadas en la comunidad del software orientado a objetos. Resumiendo, el concepto de patrón de diseño en el software proporciona una clave para ayudar a los desarrolladores a aprovechar la experiencia de otros arquitectos expertos.

En este libro, Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides introducen los principios de los patrones de diseño y ofrecen un catálogo de dichos patrones. Así, este libro realiza dos importantes contribuciones. En primer lugar, muestra el papel que los patrones pueden desempeñar diseñando la arquitectura de sistemas complejos. En segundo lugar, proporciona una referencia práctica de un conjunto de excelentes patrones que el desarrollador

puede aplicar para construir sus propias aplicaciones.

Es un honor para mí haber tenido la oportunidad de trabajar directamente con algunos de los autores de este libro en tareas de diseño arquitectónico. He aprendido mucho de ellos y sospecho que, leyendo este libro, usted también lo hará.

Grady Booch
Científico Jefe, Rational Software Corporation



A veces —en informática al menos— se da el caso de libros que, por la especial contribución que realizan en un campo determinado, se convierten en clásicos apenas han sido publicados. Este libro constituye uno de esos casos excepcionales. En efecto, a pesar de los pocos años transcurridos desde la publicación del original en inglés. *Design Patterns. Elements of Reusable Object-Oriented Software*, también conocido como GoF (de *gang of four*, o “banda de los cuatro”, en alusión a sus autores), ha revolucionado el campo de la arquitectura del software, llevando la tecnología de la orientación a objetos a un estadio más avanzado de su evolución. Hoy día, pocos son los arquitectos de software que no han oído hablar de patrones de diseño, y su uso generalmente diferencia a un buen diseño de otro que, en el mejor de los casos, resuelve un problema concreto pero que se adapta muy mal —o no lo hace en absoluto— a nuevos requisitos o cambios en los ya existentes.

No obstante, también es cierto que los patrones de diseño aún no son todo lo conocidos que debieran, especialmente por parte de los alumnos universitarios y de los recién titulados. A la rigidez de los planes de estudio, es muy posible que se una la ausencia casi total de publicaciones en español acerca de este tema. En este sentido, creemos que esta traducción al castellano puede contribuir a catapultar definitivamente este libro al lugar que se merece dentro de la comunidad hispana de la ingeniería del software. Además de servir de ayuda a los profesionales que no hayan leído el original en inglés, sin duda pasará a ser libro de texto recomendado

de la asignatura de ingeniería del software de muchas universidades, enriqueciendo así el currículum de futuras promociones de ingenieros en informática.

Por otro lado, nunca resulta sencillo traducir un libro técnico en informática. Numerosos barbarismos se han instalado ya en el idioma, de tal forma que resulta difícil a veces optar entre lo que sería el término correcto en español o el equivalente comúnmente aceptado por la comunidad de la ingeniería del software. Uno de los ejemplos más evidentes lo constituye la palabra "instancia", para designar cada uno de los objetos creados de una clase dada, y que en español tal vez debiera haberse traducido como "ejemplar". No obstante, pocas palabras hay más conocidas que ésta en la jerga de la orientación a objetos, por lo que pretender cambiarla a estas alturas habría sido un acto de vanidad por nuestra parte que sólo habría contribuido a incrementar la confusión y dificultar la comprensión del libro. En otras ocasiones, nos encontramos con palabras como *framework* o *toolkit* que hemos dejado sin traducir por el mismo motivo: el lector técnico está acostumbrado a ellas, por lo que de haberlas traducido probablemente no supiese a qué se está haciendo referencia, desvirtuando así el sentido del original.

Allá donde ha habido dudas, hemos optado por incluir ambos términos, dando preferencia al español e incluyendo el original a continuación, entre paréntesis, o bien mediante una nota al pie. Ésta ha sido la decisión adoptada para el nombre de los patrones. El nombre es parte fundamental de un patrón, ya que, una vez comprendido éste, pasa a formar parte, como los propios autores indican, de nuestro vocabulario de diseño. Por tanto, los nombres de los patrones de diseño han de ser considerados, en este sentido, como las palabras reservadas de un lenguaje de programación: no admiten traducción, so pena de engañar al lector. Sin embargo, dado que el nombre designa con una o dos palabras el propósito del patrón, parecía oportuno que el lector desconocedor de la lengua inglesa pudiese conocer su significado. Por ese motivo, hemos optado por ofrecer una traducción, entre paréntesis, al comienzo de la descripción de cada patrón, aunque posteriormente, en el texto, nos refiramos siempre a ellos con su nombre original en inglés.

Será difícil que las decisiones adoptadas sean siempre del gusto de todos, como inevitable será, a pesar de todos los esfuerzos, que

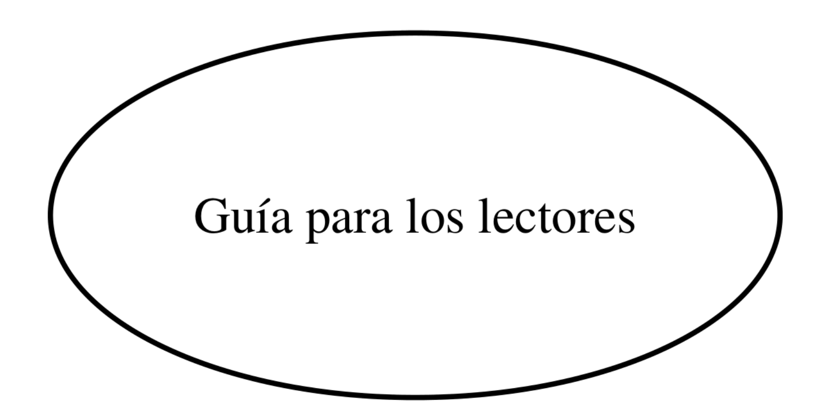
se haya escapado algún error en la versión final del libro. En cualquier caso, sepa el lector que se han puesto los mayores esfuerzos en aunar el rigor técnico con el cuidado del idioma, y apelamos a su indulgencia por los errores cometidos, de modo consciente o inconsciente.

No quisiéramos terminar este prólogo sin dar las gracias a Raúl Izquierdo Castañedo, Aquilino Adolfo Juan Fuente, Francisco Ortín Soler, José Emilio Labra Gayo y el resto de miembros del *Laboratorio de Tecnologías Orientadas a Objetos de la Universidad de Oviedo* (www.ootlab.uniovi.es), por el trabajo del grupo de investigación sobre patrones de diseño y por sus valiosos comentarios y aportaciones.

Por último, esperamos que con esta traducción al español sean muchos más los que se acerquen a los patrones de diseño, algo que sin duda contribuirá a seguir mejorando y dignificando así nuestra profesión, al hacemos mejores arquitectos de software. Igualmente, confiamos en que sirva, como ya hemos dicho, para ayudar a la formación de futuras promociones de ingenieros en informática. Haber contribuido a ello, con el minúsculo esfuerzo de una traducción, en comparación con la labor original de Eric Gamma, Richard Helm, Ralph Johnson y John Vlissides, sería algo que nos llenaría de satisfacción y orgullo.

César Fernández Acebal
Juan Manuel Cueva Lovelle

Oviedo, 20 de junio de 2002



Guía para los lectores

Este libro tiene dos partes principales. La primera parte (Capítulos 1 y 2) describe qué son los patrones de diseño y cómo pueden ayudarle a diseñar software orientado a objetos. Incluye un caso de estudio que demuestra cómo se aplican en la práctica. La segunda parte del libro (Capítulos 3, 4 y 5) es un catálogo de los patrones de diseño propiamente dichos.

El catálogo constituye la mayor parte del libro. Sus capítulos dividen los patrones de diseño en tres tipos: de creación, estructurales y de comportamiento. El catálogo se puede usar de varias formas: puede leerse de principio a fin o se puede simplemente ir de un patrón a otro. Otra posibilidad es estudiar uno de los capítulos. Esto le ayudará a ver cómo se distinguen entre sí patrones estrechamente relacionados.

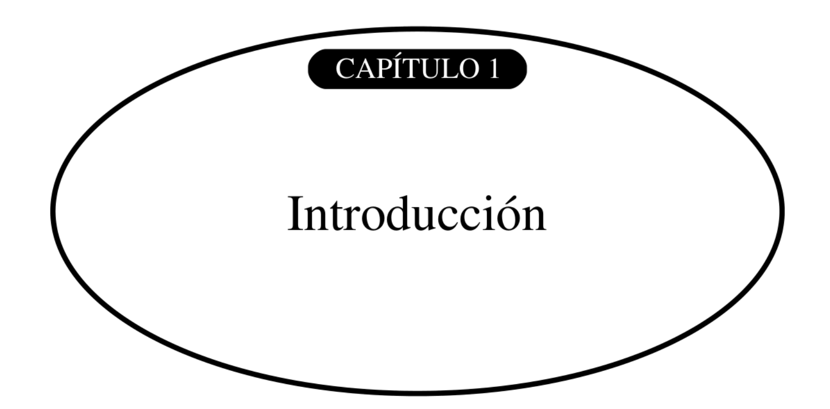
Puede usar las referencias entre los patrones como una ruta lógica a través del catálogo. Este enfoque le hará comprender cómo se relacionan los patrones entre sí, cómo se pueden combinar con otros y qué patrones funcionan bien juntos. La Figura 1.1 muestra estas referencias gráficamente.

Otra forma más de leer el catálogo es usar un enfoque más orientado al problema. Vaya directamente a la Sección 1.6 (página 10) para leer algunos problemas comunes a la hora de diseñar software orientado a objetos reutilizable; después, lea los patrones que resuelven estos problemas. Algunas personas leen primero el catálogo completo y *luego* usan un enfoque orientado al problema para aplicar los patrones en sus proyectos.

Si no es un diseñador orientado a objetos experimentado, le recomendamos que empiece con los patrones más sencillos y más comunes:

- Abstract Factory (página 79)
- Adapter (131)
- Composite (151)
- Decorator (161)
- Factory Method (99)
- Observer (269)
- Strategy (289)
- Template Method (299)

Es difícil encontrar un sistema orientado a objetos que no use al menos un par de estos patrones, y los grandes sistemas los usan casi todos. Este subconjunto le ayudará a entender los patrones de diseño en particular y el buen diseño orientado a objetos en general.



CONTENIDO DEL CAPÍTULO

Diseñar software orientado a objetos es difícil, y aún lo es más diseñar software orientado a objetos reutilizable. Hay que encontrar los objetos pertinentes, factorizarlos en clases con la granularidad adecuada, definir interfaces de clases y jerarquías de herencia y establecer las principales relaciones entre esas clases y objetos. Nuestro diseño debe ser específico del problema que estamos manejando, pero también lo suficientemente general para adecuarse a futuros requisitos y problemas. También queremos evitar el rediseño, o al menos minimizarlo. Los diseñadores experimentados de software orientado a objetos nos dirán que es difícil, sino imposible, lograr un diseño flexible y reutilizable a la primera y que, antes de terminar un diseño, es frecuente intentar reutilizarlo varias veces, modificándolo cada una de ellas.

Sin embargo, estos diseñadores experimentados realmente consiguen hacer buenos diseños, mientras que los diseñadores novatos se ven abrumados por las opciones disponibles y tienden a recurrir a las técnicas no orientadas a objetos que ya usaron antes. A un principiante le lleva bastante tiempo aprender en qué consiste un buen diseño orientado a objetos. Es evidente que los diseñadores experimentados saben algo que los principiantes no. ¿Qué es?

Algo que los expertos saben que *no* hay que hacer es resolver cada problema partiendo de cero. Por el contrario, reutilizan soluciones que ya les han sido útiles en el pasado. Cuando encuentran una solución buena, la usan una y otra vez. Esa experiencia es parte de lo que les convierte en expertos. Por tanto, nos encontraremos con patrones recurrentes de clases y comunicaciones entre objetos en muchos sistemas orientados a objetos. Estos patrones resuelven problemas concretos de diseño y hacen que los diseños orientados a objetos sean más flexibles, elegantes y reutilizables. Los patrones ayudan a los diseñadores a reutilizar buenos diseños al basar los nuevos diseños en la experiencia previa. Un diseñador familiarizado con dichos patrones

puede aplicarlos inmediatamente en los problemas de diseño sin tener que redescubrirlos.

Ilustremos este punto con una analogía. Los novelistas y escritores rara vez diseñan las tramas de sus obras desde cero, sino que siguen patrones como el del *héroe trágico* (Macbeth, Hamlet, etc.) o *la novela romántica* (innumerables novelas de amor). Del mismo modo, los diseñadores orientados a objetos siguen patrones como “representar estados con objetos” o “decorar objetos de manera que se puedan añadir y borrar funcionalidades fácilmente”. Una vez que conocemos el patrón, hay muchas decisiones de diseño que se derivan de manera natural.

Todos sabemos el valor de la experiencia en el diseño. ¿Cuántas veces hemos tenido un *déjá-vu* de diseño —esa sensación de que ya hemos resuelto ese problema antes, pero no sabemos exactamente dónde ni cómo—? Si pudiéramos recordar los detalles del problema anterior y de cómo lo resolvimos podríamos valernos de esa experiencia sin tener que reinventar la solución. Sin embargo, no solemos dedicamos a dejar constancia de nuestra experiencia en el diseño de software para que la usen otros.

El propósito de este libro es documentar la experiencia en el diseño de software orientado a objetos en forma de **patrones de diseño**. Cada patrón nomina, explica y evalúa un diseño importante y recurrente en los sistemas orientados a objetos. Nuestro objetivo es representar esa experiencia de diseño de forma que pueda ser reutilizada de manera efectiva por otras personas. Para lograrlo, hemos documentado algunos de los patrones de diseño más importantes y los presentamos como un catálogo.

Los patrones de diseño hacen que sea más fácil reutilizar buenos diseños y arquitecturas. Al expresar como patrones de diseño técnicas que ya han sido probadas, las estamos haciendo más accesibles para los desarrolladores de nuevos sistemas. Los patrones de diseño nos ayudan a elegir las alternativas de diseño que hacen que un sistema sea reutilizable, y a evitar aquéllas que dificultan dicha reutilización. Pueden incluso mejorar la documentación y el mantenimiento de los sistemas existentes al proporcionar una especificación explícita de las interacciones entre clases y objetos y

de cuál es su intención. En definitiva, los patrones de diseño ayudan a un diseñador a lograr un buen diseño más rápidamente.

Ninguno de los patrones de diseño de este libro describe diseños nuevos o que no hayan sido probados. Se han incluido sólo aquellos que se han aplicado más de una vez en diferentes sistemas. La mayoría de ellos estaba sin documentar, y existían bien como parte del repertorio de la comunidad de la orientación a objetos, bien como elementos de algunos buenos sistemas orientados a objetos — y de ninguna de ambas formas resultan fáciles de aprender por los diseñadores novatos—. Así que, aunque estos diseños no son nuevos, los hemos expresado de una forma nueva y accesible: como un catálogo de patrones de diseño que tienen un formato consistente.

A pesar del tamaño del libro, los patrones de diseño que hay en él representan sólo una parte de lo que puede saber un experto. No contiene patrones que tengan que ver con concurrencia o programación distribuida o en tiempo real. Tampoco hay patrones de dominios específicos. No se cuenta cómo construir interfaces de usuario, cómo escribir controladores de dispositivos o cómo usar una base de datos orientada a objetos. Cada una de estas áreas tiene sus propios patrones, y sería bueno que alguien los catalogase también.

1.1. ¿QUÉ ES UN PATRÓN DE DISEÑO?

Según Christopher Alexander, “cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema, de tal modo que se pueda aplicar esta solución un millón de veces, sin hacer lo mismo dos veces” [AIS+77, página xi. Aunque Alexander se refería a patrones en ciudades y edificios, lo que dice también es válido para patrones de diseño orientados a objetos. Nuestras soluciones se expresan en términos de objetos e interfaces, en vez de paredes y puertas, pero en la esencia de ambos tipos de patrones se encuentra una solución a un problema dentro de un contexto.

En general, un patrón tiene cuatro elementos esenciales:

1. El **nombre del patrón** permite describir, en una o dos palabras, un problema de diseño junto con sus soluciones y consecuencias. Al dar nombre a un patrón inmediatamente estamos incrementado nuestro vocabulario de diseño, lo que nos permite diseñar con mayor abstracción. Tener un vocabulario de patrones nos permite hablar de ellos con otros colegas, mencionarlos en nuestra documentación y tenerlos nosotros mismos en cuenta. De esta manera, resulta más fácil pensar en nuestros diseños y transmitirlos a otros, junto con sus ventajas e inconvenientes. Encontrar buenos nombres ha sido una de las partes más difíciles al desarrollar nuestro catálogo.

2. El **problema** describe cuándo aplicar el patrón. Explica el problema y su contexto. Puede describir problemas concretos de diseño (por ejemplo, cómo representar algoritmos como objetos), así como las estructuras de clases u objetos que son sintomáticas de un diseño inflexible. A veces el problema incluye una serie de condiciones que deben darse para que tenga sentido aplicar el patrón.

3. La **solución** describe los elementos que constituyen el diseño, sus relaciones, responsabilidades y colaboraciones. La solución no describe un diseño o una implementación en concreto, sino que un patrón es más bien como una plantilla que puede aplicarse en muchas situaciones diferentes. El patrón proporciona una descripción abstracta de un problema de diseño y cómo lo resuelve una disposición general de elementos (en nuestro caso, clases y objetos).

4. Las **consecuencias** son los resultados así como las ventajas e inconvenientes de aplicar el patrón. Aunque cuando se describen decisiones de diseño muchas veces no se reflejan sus consecuencias, éstas son fundamentales para evaluar las alternativas de diseño y comprender los costes y beneficios de aplicar el patrón. Las consecuencias en el software suelen referirse al equilibrio entre espacio y

> tiempo. También pueden tratar cuestiones de lenguaje e implementación. Por otro lado, puesto que la reutilización suele ser uno de lo factores de los diseños orientados a objetos, las consecuencias de un patrón incluyen su impacto sobre la flexibilidad, extensibilidad y portabilidad de un sistema. Incluir estas consecuencias de un modo explícito nos ayudará a comprenderlas y evaluarlas.

Qué es y qué no es un patrón de diseño es una cuestión que depende del punto de vista de cada uno. Lo que para una persona es un patrón puede ser un bloque primitivo de construcción para otra. En este libro nos hemos centrado en patrones situados en un cierto nivel de abstracción. *Patrones de Diseño* no se ocupa de diseños como listas enlazadas y tablas de dispersión (*hash*) que pueden codificarse en clases y reutilizarse como tales. Tampoco se trata de complicados diseños específicos de un dominio para una aplicación o subsistema completo. Los patrones de diseño de este libro son *descripciones de clases y objetos relacionados que están particularizados para resolver un problema de diseño general en un determinado contexto*.

Un patrón de diseño nomina, abstrae e identifica los aspectos clave de una estructura de diseño común, lo que los hace útiles para crear un diseño orientado a objetos reutilizable. El patrón de diseño identifica las clases e instancias participantes, sus roles y colaboraciones, y la distribución de responsabilidades. Cada patrón de diseño se centra en un problema concreto, describiendo cuándo aplicarlo y si tiene sentido hacerlo teniendo en cuenta otras restricciones de diseño, así como las consecuencias y las ventajas e inconvenientes de su uso. Por otro lado, como normalmente tendremos que implementar nuestros diseños, un patrón también proporciona código de ejemplo en C++, y a veces en Smalltalk, para ilustrar una implementación.

Aunque los patrones describen diseños orientados a objetos, están basados en soluciones prácticas que han sido implementadas en los lenguajes de programación orientados a objetos más usuales, como Smalltalk y C++, en vez de mediante lenguajes procedimentales (Pascal, C, Ada) u otros lenguajes orientados a objetos más dinámicos (CLOS, Dylan, Self). Nosotros hemos elegido

Smalltalk y C++ por una cuestión pragmática: nuestra experiencia diaria ha sido con estos lenguajes, y éstos cada vez son más populares.

La elección del lenguaje de programación es importante, ya que influye en el punto de vista. Nuestros patrones presuponen características de los lenguajes Smalltalk y C++, y esa elección determina lo que puede implementarse o no fácilmente. Si hubiéramos supuesto lenguajes procedimentales, tal vez hubiéramos incluido patrones llamados "Herencia", "Encapsulación" y "Polimorfismo". De manera similar, algunos de nuestros patrones están incluidos directamente en lenguajes orientados a objetos menos corrientes. CLOS, por ejemplo, tiene multi-métodos que reducen la necesidad de patrones como el Visitor (página 305). De hecho, hay suficientes diferencias entre Smalltalk y C++ como para que algunos patrones puedan expresarse más fácilmente en un lenguaje que en otro (por ejemplo, el Iterator (237)).

1.2. PATRONES DE DISEÑO EN EL MVC DE SMALLTALK

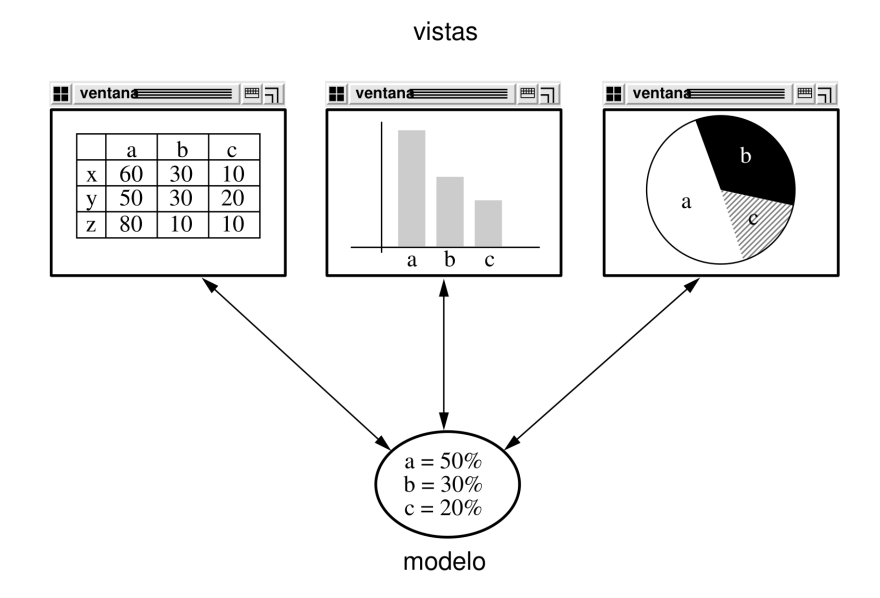
La tríada de clases Modelo/Vista/Controlador (MVC) [KP88] se usa para construir interfaces de usuario en Smalltalk-80. Observar los patrones de diseño que hay en MVC debería ayudar a entender qué queremos decir con el término "patrón".

MVC consiste en tres tipos de objetos. El Modelo es el objeto de aplicación, la Vista es su representación en pantalla y el Controlador define el modo en que la interfaz reacciona a la entrada del usuario. Antes de MVC, los diseños de interfaces de usuario tendían a agrupar estos objetos en uno solo. MVC los separa para incrementar la flexibilidad y reutilización.

MVC desacopla las vistas de los modelos estableciendo entre ellos un protocolo de suscripción/notificación. Una vista debe asegurarse de que su apariencia refleja el estado del modelo. Cada vez que cambian los datos del modelo, éste se encarga de avisar a las vistas que dependen de él. Como respuesta a dicha notificación, cada vista tiene la oportunidad de actualizarse a sí misma. Este

enfoque permite asignar varias vistas a un modelo para ofrecer diferentes presentaciones. También se pueden crear nuevas vistas de un modelo sin necesidad de volver a escribir éste.

El siguiente diagrama muestra un modelo y tres vistas (hemos dejado fuera los controladores para simplificar). El modelo contiene algunos valores de datos y las vistas, consistentes en una hoja de cálculo, un histograma y un gráfico de tarta que muestran estos datos de varias formas. El modelo se comunica con sus vistas cuando cambian sus valores, y las vistas se comunican con el modelo para acceder a éstos.



Si nos fiamos de él, este ejemplo refleja un diseño que desacopla las vistas de los modelos. Pero el diseño es aplicable a un problema más general: desacoplar objetos de manera que los cambios en uno puedan afectar a otros sin necesidad de que el objeto que cambia conozca detalles de los otros. Dicho diseño más general se describe en el patrón Observer (página 269).

Otra característica de MVC es que las vistas se pueden anidar. Por ejemplo, un panel de control con botones se puede implementar como una vista compleja que contiene varias vistas de botones anidadas. La interfaz de usuario de un inspector de objetos puede consistir en vistas anidadas que pueden ser reutilizadas en un

depurador. MVC permite vistas anidadas gracias a la clase VistaCompuesta, una subclase de Vista. Los objetos VistaCompuesta pueden actuar simplemente como objetos Vista, es decir, una vista compuesta puede usarse en cualquier lugar donde pudiera usarse una vista, pero también contiene y gestiona vistas anidadas.

De nuevo, podríamos pensar en él como un diseño que nos permite tratar a una vista compuesta exactamente igual que a uno de sus componentes. Pero el diseño es aplicable a un problema más general, que ocurre cada vez que queremos agrupar objetos y tratar al grupo como a un objeto individual. El patrón Composite (151) describe este diseño más general. Dicho patrón permite crear una jerarquía en la que algunas subclases definen objetos primitivos (por ejemplo, Boton) y otras, objetos compuestos (VistaCompuesta), que ensamblan los objetos primitivos en otros más complejos.

MVC también permite cambiar el modo en que una vista responde a la entrada de usuario sin cambiar su representación visual. En este sentido, tal vez queramos cambiar cómo responde al teclado, por ejemplo, o hacer que use un menú contextual en vez de atajos de teclado. MVC encapsula el mecanismo de respuesta en un objeto Controlador. Hay una jerarquía de controladores y es fácil crear un nuevo controlador como una variación de uno existente.

Una vista usa una instancia de una subclase de Controlador para implementar una determinada estrategia de respuesta; para implementar una estrategia diferente, simplemente basta con sustituir la instancia por otra clase de controlador. Incluso es posible cambiar el controlador de una vista en tiempo de ejecución, para hacer que la vista cambie el modo en que responde a la entrada de usuario. Por ejemplo, para desactivar una vista y que no acepte entradas basta con asignarle un controlador que haga caso omiso de los eventos de entrada.

La relación entre Vista y Controlador es un ejemplo del patrón Strategy (289). Una Estrategia es un objeto que representa un algoritmo. Es útil cuando queremos reemplazar el algoritmo, ya sea estática o dinámicamente, cuando existen muchas variantes del mismo o cuando tiene estructuras de datos complejas que queremos encapsular.

MVC usa otros patrones de diseño, tales como el Factory Method (99) para especificar la clase controlador predeterminada de una

vista, y el Decorator (161) para añadir capacidad de desplazamiento a una vista. Pero las principales relaciones en MVC se dan entre los patrones de diseño Observer. Composite y Strategy.

1.3. DESCRIPCIÓN DE LOS PATRONES DE DISEÑO

¿Cómo describimos los patrones de diseño? Las notaciones gráficas, aunque importantes, no son suficientes. Simplemente representan el producto final del proceso de diseño, como las relaciones entre clases y objetos. Para reutilizar el diseño, debemos hacer constar las decisiones, alternativas y ventajas e inconvenientes que dieron lugar a él. También son importantes los ejemplos concretos, porque nos ayudan a ver el diseño en acción.

Describimos los patrones de diseño empleando un formato consistente. Cada patrón se divide en secciones de acuerdo a la siguiente plantilla. Ésta da una estructura uniforme a la información, haciendo que los patrones de diseño sean más fáciles de aprender, comparar y usar.

Nombre del patrón y clasificación

El nombre del patrón transmite sucintamente su esencia. Un buen nombre es vital, porque pasará a formar parte de nuestro vocabulario de diseño. La clasificación del patrón refleja el esquema que se presenta en la Sección 1.5.

Propósito

Una frase breve que responde a las siguientes cuestiones: ¿Qué hace este patrón de diseño? ¿En qué se basa? ¿Cuál es el problema concreto de diseño que resuelve?

También conocido como

Otros nombres, si existen, por los que se conoce al patrón.

Motivación

Un escenario que ilustra un problema de diseño y cómo las

estructuras de clases y objetos del patrón resuelven el problema. El escenario ayudará a entender la descripción que sigue.

Aplicabilidad

¿En qué situaciones se puede aplicar el patrón de diseño? ¿Qué ejemplos hay de malos diseños que el patrón puede resolver? ¿Cómo se puede reconocer dichas situaciones?

Estructura

Una representación gráfica de las clases del patrón usando una notación basada en la Técnica de Modelado de Objetos (OMT) [RBP+9I]. También hacemos uso de diagramas de interacción (JCJ092, Boo94) para mostrar secuencias de peticiones y colaboraciones entre objetos. El Apéndice B describe estas notaciones en detalle.

Participantes

Las clases y objetos participantes en el patrón de diseño, junto con sus responsabilidades.

Colaboraciones

Cómo colaboran los participantes para llevar a cabo sus responsabilidades.

Consecuencias

¿Cómo consigue el patrón sus objetivos? ¿Cuáles son las ventajas e inconvenientes y los resultados de usar el patrón? ¿Qué aspectos de la estructura del sistema se pueden modificar de forma independiente?

Implementación

¿Cuáles son las dificultades, trucos o técnicas que deberíamos tener en cuenta a la hora de aplicar el patrón? ¿Hay cuestiones específicas del lenguaje?

Código de ejemplo

Fragmentos de código que muestran cómo se puede implementar el patrón en C++ o en Smalltalk.

Usos conocidos

Ejemplos del patrón en sistemas reales. Incluimos al menos dos ejemplos de diferentes dominios.

Patrones relacionados

¿Qué patrones de diseño están estrechamente relacionados con éste? ¿Cuáles son las principales diferencias? ¿Con qué otros patrones debería usarse?

Los apéndices proporcionan información adicional que le ayudará a entender los patrones y las discusiones que los rodean. El Apéndice A es un glosario de la terminología empleada en el libro. En el Apéndice B que ya hemos mencionado se presentan las diferentes notaciones. También describiremos aspectos de las notaciones a medida que las vayamos introduciendo en las discusiones venideras. Finalmente, el Apéndice C contiene el código fuente de las clases básicas que usamos en los ejemplos.

1.4. EL CATÁLOGO DE PATRONES DE DISEÑO

El catálogo que comienza en la página 71 contiene 23 patrones de diseño. A continuación mostraremos el nombre y el propósito de cada uno de ellos para ofrecerle una perspectiva general. El número entre paréntesis que sigue a cada patrón representa el número de página de éste (una convención que seguiremos a lo largo de todo el libro)[1].

***Abstract Factory* (Fábrica Abstracta) (79)**

Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.

***Adapter* (Adaptador) (131)**

Convierte la interfaz de una clase en otra distinta que es la

que esperan los clientes. Permite que cooperen clases que de otra manera no podrían por tener interfaces incompatibles.

Bridge **(Puente) (141)**

Desacopla una abstracción de su implementación, de manera que ambas puedan variar de forma independiente.

Builder **(Constructor) (89)**

Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.

Chain of Responsibility **(Cadena de Responsabilidad) (205)**

Evita acoplar el emisor de una petición a su receptor, al dar a más de un objeto la posibilidad de responder a la petición. Crea una cadena con los objetos receptores y pasa la petición a través de la cadena hasta que ésta sea tratada por algún objeto.

Command **(Orden) (215)**

Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con distintas peticiones, encolar o llevar un registro de las peticiones y poder deshacer las operaciones.

Composite **(Compuesto) (151)**

Combina objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.

Decorator **(Decorador) (161)**

Añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

Facade **(Fachada) (171)**

Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel

que hace que el subsistema sea más fácil de usar.

Factory Method **(Método de Fabricación) (99)**

Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instancias Permite que una clase delegue en sus subclases la creación de objetos.

Flyweight **(Peso Ligero)[2] (179)**

Usa el compartimiento para permitir un gran número de objetos de grano fino de forma eficiente.

Interpreter **(Intérprete) (225)**

Dado un lenguaje, define una representación de su gramática junto con un intérprete que usa dicha representación para interpretar sentencias del lenguaje.

Iterator **(Iterador) (237)**

Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

Mediator **(Mediador) (251)**

Define un objeto que encapsula cómo interactúan un conjunto de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.

Memento **(Recuerdo) (261)**

Representa y externaliza el estado interno de un objeto sin violar la encapsulación, de forma que éste puede volver a dicho estado más tarde.

Observer **(Observador) (269)**

Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifica y se actualizan automáticamente todos los objetos que dependen de él.

Prototype **(Prototipo) (109)**

Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando de este prototipo.

***Proxy* (Apoderado) (191)**

Proporciona un sustituto o representante de otro objeto para controlar el acceso a éste.

***Singleton* (Único) (119)**

Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.

***State* (Estado) (279)**

Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.

***Strategy* (Estrategia) (289)**

Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.

***Template Method* (Método Plantilla) (299)**

Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura. *Visitor* (Visitante) (305)

Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

1.5. ORGANIZACIÓN DEL CATÁLOGO

Los patrones de diseño varían en su granularidad y nivel de abstracción. Dado que existen muchos patrones de diseño, es necesario un modo de organizados. Esta sección clasifica los

patrones de diseño de manera que podamos referirnos a familias de patrones relacionados. La clasificación ayuda a aprender más rápidamente los patrones del catálogo, y también puede encauzar los esfuerzos para descubrir nuevos patrones.

Tabla 1.1: Patrones de diseño

		Propósito		
		De Creación	**Estructurales**	**De comportamiento**
Ámbito	**Clase**	Factory Method (99)	Adapter (de clases) (131)	Interpreter (225) Template Method (299)
	Objeto	Abstract Factory (79) Builder (89) Prototype (109) Singleton (119)	Adapter (de objetos) (131) Bridge (141) Composite (151) Decorator (161) Facade (171) Flyweight (179) Proxy (191)	Chain of Responsibility (205) Command (215) Iterator (237) Mediator (251) Memento (261) Observer (269) State (279) Strategy (289) Visitor (305)

Nosotros clasificamos los patrones de diseño siguiendo dos criterios (Tabla 1.1). El primero de ellos, denominado **propósito**, refleja qué hace un patrón. Los patrones pueden tener un propósito **de creación, estructural** o **de comportamiento**. Los patrones de creación tienen que ver con el proceso de creación de objetos. Los patrones estructurales tratan con la composición de clases u objetos. Los de comportamiento caracterizan el modo en que las clases y objetos interactúan y se reparten la responsabilidad.

El segundo criterio, denominado **ámbito**, especifica si el patrón se aplica principalmente a clases o a objetos. Los patrones de clases se ocupan de las relaciones entre las clases y sus subclases. Estas relaciones se establecen a través de la herencia, de modo que son relaciones estáticas —fijadas en tiempo de compilación—. Los patrones de objetos tratan con las relaciones entre objetos, que pueden cambiarse en tiempo de ejecución y son más dinámicas. Casi todos los patrones usan la herencia de un modo u otro, así que los únicos patrones etiquetados como “patrones de clases” son

aquellos que se centrar en las relaciones entre clases. Nótese que la mayoría de los patrones tienen un ámbito de objeto.

Los patrones de creación de clases delegan alguna parte del proceso de creación de objetos en las subclases, mientras que los patrones de creación de objetos lo hacen en otro objeto. Los patrones estructurales de clases usan la herencia para componer clases, mientras que los de objetos describen formas de ensamblar objetos. Los patrones de comportamiento de clases usan la herencia para describir algoritmos y flujos de control, mientras que los de objetos describen cómo cooperan un grupo de objetos para realizar una tarea que ningún objeto puede llevar a cabo por sí solo.

Hay otras maneras de organizar los patrones. Algunos patrones suelen usarse juntos. Por ejemplo, el Composite suele usarse con el Iterator o el Visitor. Algunos patrones son alternativas: el Prototype es muchas veces una alternativa al Abstract Factory. Algunos patrones dan como resultado diseños parecidos, a pesar de que tengan diferentes propósitos. Por ejemplo, los diagramas de estructura del Composite y el Decorator son similares.

Otro modo de organizar los patrones de diseño es en función de cómo se hagan referencia unos a otros en su sección “Patrones Relacionados”. La Figura 1.1 representa estas relaciones gráficamente.

Es evidente que hay muchas formas de organizar los patrones de diseño. Tener muchas formas de pensar en los patrones le hará comprender mejor qué es lo que hacen, cómo compararlos y cuándo aplicarlos.

1.6. CÓMO RESUELVEN LOS PATRONES LOS PROBLEMAS DE DISEÑO

Los patrones de diseño resuelven muchos de los problemas diarios con los que se enfrentan los diseñadores orientados a objetos, y lo hacen de muchas formas diferentes. A continuación se muestran algunos de estos problemas y cómo los solucionan los patrones.

ENCONTRAR LOS OBJETOS APROPIADOS

Los programas orientados a objetos están formados de objetos. Un **objeto** encapsula tanto datos como los procedimientos que operan sobre esos datos. Estos procedimientos es lo que se conoce normalmente como **métodos u operaciones**. Un objeto realiza una operación cuando recibe una **petición** (o **mensaje**) de un **cliente**.

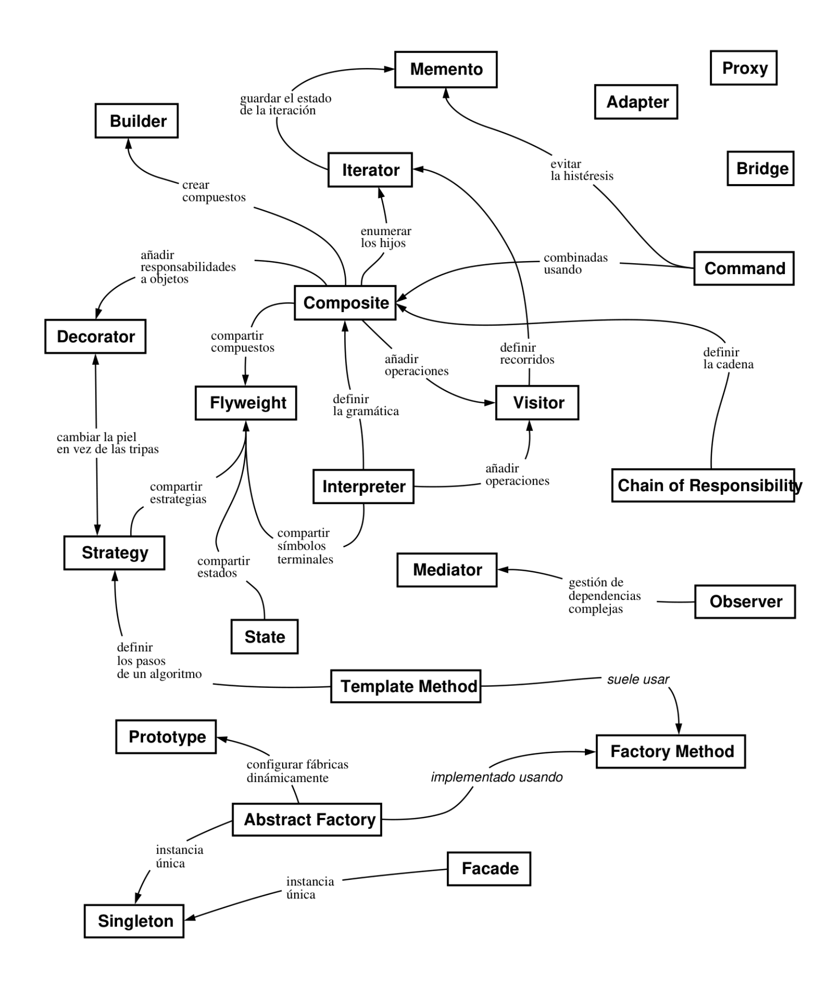
Las peticiones son el *único* modo de lograr que un objeto ejecute una operación. Las operaciones son la *única* forma de cambiar los datos internos de un objeto. Debido a estas restricciones, se dice que el estado interno de un objeto está **encapsulado**; no puede accederse a él directamente, y su representación no es visible desde el exterior del objeto.

Lo más complicado del diseño orientado a objetos es descomponer un sistema en objetos. La tarea es difícil porque entran en juego muchos factores: encapsulación, granularidad, dependencia, flexibilidad, rendimiento, evolución, reutilización, etcétera, etcétera. Todos ellos influyen en la descomposición, muchas veces de formas opuestas.

Las metodologías orientadas a objetos permiten muchos enfoques diferentes. Podemos escribir la descripción de un problema, extraer los nombres y verbos, y crear las correspondientes clases y operaciones. O podemos centrarnos en las colaboraciones y responsabilidades de nuestro sistema. O modelar el mundo real y traducir al diseño los objetos encontrados durante el análisis. Siempre habrá discrepancias sobre qué enfoque es mejor.

Muchos objetos de un diseño proceden del modelo del análisis. Pero los diseños orientados a objetos suelen acabar teniendo clases que no tienen su equivalente en el mundo real. Algunas de ellas son clases de bajo nivel como los arrays. Otras son de mucho más alto nivel. Por ejemplo, el patrón Composite (151) introduce una abstracción para tratar de manera uniforme objetos que no tienen un equivalente físico. El modelado estricto del mundo real conduce a un sistema que refleja las realidades presentes pero no necesariamente las futuras. Las abstracciones que surgen durante el diseño son fundamentales para lograr un diseño flexible.

Figura 1.1: Relaciones entre los patrones de diseño



Los patrones de diseño ayudan a identificar abstracciones menos obvias y los objetos que las expresan. Por ejemplo, los objetos que representan un proceso o algoritmo no tienen lugar en la naturaleza, y sin embargo son una parte crucial de los diseños flexibles. El patrón Strategy (289) describe cómo implementar familias intercambiables de algoritmos. El patrón State (279) representa cada estado de una entidad como un objeto. Estos objetos rara vez se encuentran durante el análisis o incluso en las primeras etapas del diseño; son descubiertos más tarde, mientras se trata de hacer al diseño más flexible y reutilizable.

DETERMINAR LA GRANULARIDAD DE LOS OBJETOS

Los objetos pueden variar enormemente en tamaño y número. Pueden representar cualquier cosa, desde el hardware hasta aplicaciones completas. ¿Cómo decidir entonces qué debería ser un objeto?

Los patrones de diseño también se encargan de esta cuestión. El patrón Facade (171) describe cómo representar subsistemas completos como objetos, y el patrón Flyweight (179) cómo permitir un gran número de objetos de granularidad muy fina. Otros patrones de diseño describen formas concretas de descomponer un objeto en otros más pequeños. Los patrones Abstract Factory (79) y Builder (89) producen objetos cuya única responsabilidad es crear otros objetos. El patrón Visitor (305) y el Command (215) dan lugar a objetos cuya única responsabilidad es implementar una petición en otro objeto o grupo de objetos.

ESPECIFICAR LAS INTERFACES DE LOS OBJETOS

Cada operación declarada por un objeto especifica el nombre de la operación, los objetos que toma como parámetros y el valor de retomo de la operación. Esto es lo que se conoce como la **signatura** de la operación. Al conjunto de todas las signaturas definidas por las operaciones de un objeto se le denomina la **interfaz** del objeto. Dicha interfaz caracteriza al conjunto completo de peticiones que se pueden enviar al objeto. Cualquier petición que concuerde con una signatura de la interfaz puede ser enviada al objeto.

Un **tipo** es un nombre que se usa para denotar una determinada interfaz. Decimos que un objeto tiene el tipo “Ventana” si acepta todas las peticiones definidas en una interfaz llamada “Ventana”. Un objeto puede tener muchos tipos, y objetos muy diferentes pueden compartir un mismo tipo. Parte de la interfaz de un objeto puede ser caracterizada por un tipo, y otras partes por otros tipos. Dos objetos del mismo tipo sólo necesitan compartir partes de sus interfaces. Las interfaces pueden contener, como subconjuntos, otras interfaces. Se dice que un tipo es un **subtipo** de otro si su interfaz contiene a la interfaz de su **supertipo**. Suele decirse que un subtipo *hereda* la interfaz de su supertipo.

Las interfaces son fundamentales en los sistemas orientados a

objetos. Los objetos sólo se conocen a través de su interfaz. No hay modo de saber nada de un objeto o pedirle que haga nada si no es a través de su interfaz. La interfaz de un objeto no dice nada acerca de su implementación —distintos objetos son libres de implementar las peticiones de forma diferente—. Eso significa que dos objetos con implementaciones completamente diferentes pueden tener interfaces idénticas.

Cuando se envía una petición a un objeto, la operación concreta que se ejecuta depende *tanto* de la petición *como* del objeto que la recibe. Objetos diferentes que soportan peticiones idénticas pueden tener distintas implementaciones de las operaciones que satisfacen esas peticiones. La asociación en tiempo de ejecución entre una petición a un objeto y una de sus operaciones es lo que se conoce como enlace **dinámico**.

El enlace dinámico significa que enviar una petición no nos liga a una implementación particular hasta el tiempo de ejecución. Por tanto, podemos escribir programas que esperen un objeto con una determinada interfaz, sabiendo que cualquier objeto que tenga la interfaz correcta aceptará la petición.

Más aún, el enlace dinámico nos permite sustituir objetos en tiempo de ejecución por otros que tengan la misma interfaz. Esta capacidad de sustitución es lo que se conoce como polimorfismo, y es un concepto clave en los sistemas orientados a objetos. Permite que un cliente haga pocas suposiciones sobre otros objetos aparte de que permitan una interfaz determinada. El polimorfismo simplifica las definiciones de los clientes, desacopla unos objetos de otros y permite que varíen las relaciones entre ellos en tiempo de ejecución.

Los patrones de diseño ayudan a definir interfaces identificando sus elementos clave y los tipos de datos que se envían a la interfaz. Un patrón de diseño también puede decir qué *no* debemos poner en la interfaz. El patrón Memento (Recuerdo) (261) es un buen ejemplo de esto. Dicho patrón describe cómo encapsular y guardar el estado interno de un objeto para que éste pueda volver a ese estado posteriormente. El patrón estipula que los objetos Recuerdo deben definir dos interfaces: una restringida, que permita a los clientes albergar y copiar el estado a recordar, y otra protegida que sólo pueda usar el objeto original para almacenar y recuperar dicho

estado.

Los patrones de diseño también especifican relaciones entre interfaces. En concreto, muchas veces requieren que algunas clases tengan interfaces parecidas, o imponen restricciones a las interfaces de algunas clases. Por ejemplo, tanto el patrón Decorator (161) como el Proxy (191) requieren que las interfaces de los objetos Decorador y Proxy sean idénticos a los objetos decorado y representado, respectivamente. En el patrón Visitor (305), la interfaz Visitante debe reflejar todas las clases de objetos que pueden ser visitados.

ESPECIFICAR LAS IMPLEMENTACIONES DE LOS OBJETOS

Hasta ahora hemos dicho poco sobre cómo definir realmente un objeto. La implementación de un objeto queda definida por su clase. La clase especifica los datos, la representación interna del objeto y define las operaciones que puede realizar.

Nuestra notación basada en OMT (resumida en el Apéndice B) muestra una clase como un rectángulo con el nombre en negrita. Las operaciones aparecen en un tipo de fuente normal bajo el nombre de la clase. Cualquier dato que defina la clase viene después de las operaciones. Por último, se utilizan líneas para separar el nombre de la clase de las operaciones y a éstas de los datos:

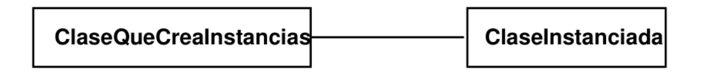
NombreDeLaClase
Operacion1() Tipo Operacion2() ...
variableDeInstancia1 Tipo variableDeInstancia2 ...

Los tipos de retomo y de las variables de instancia son opcionales, puesto que no suponemos un lenguaje de implementación estáticamente *tipado*.

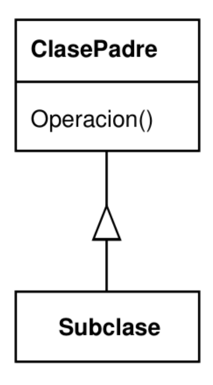
Los objetos se crean instanciando una clase. Se dice que el objeto es una instancia de la clase. El proceso de crear una instancia de una clase asigna espacio de almacenamiento para los datos

internos del objeto (representados por variables de instancia) y asocia las operaciones con esos datos. Se pueden crear muchas instancias parecidas de un objeto instanciando una clase.

Una flecha de línea discontinua indica que una clase crea objetos de otra clase. La flecha apunta a la clase de los objetos creados.



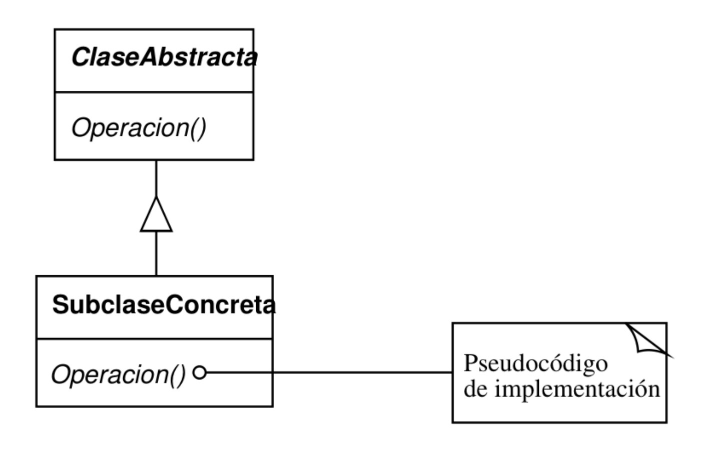
Las nuevas clases se pueden definir en términos de las existentes, usando la **herencia de clases**. Cuando una **subclase** hereda de una **clase padre**, incluye las definiciones de todos los datos y operaciones que define la clase padre. Los objetos que son instancias de las subclases contendrán todos los datos definidos por la subclase y por sus clases padre, y serán capaces de realizar todas las operaciones definidas por sus subclases y sus padres. Indicamos la relación de subclase con una línea vertical y un triángulo:



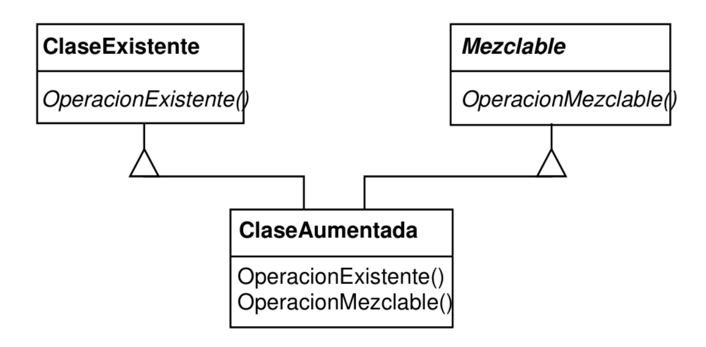
Una **clase abstracta** es aquélla cuyo propósito principal es definir una interfaz común **para** sus subclases. Una clase abstracta delegará parte o toda su implementación en las operaciones definidas **en** sus subclases; de ahí que no se pueda crear una instancia de una clase abstracta. Las **operaciones** que una clase abstracta define pero no implementa se denominan operaciones **abstractas. Las** clases que no son abstractas se denominan **clases** concretas.

Las subclases pueden redefinir el comportamiento de sus clases padres. Más concretamente, una clase puede redefinir una operación definida por su clase padre. La redefinición permite que sean las subclases las que manejen una petición en vez de sus clases padres. La herencia de clases permite definir clases simplemente extendiendo otras clases, haciendo que sea fácil definir familias de objetos de funcionalidad parecida.

Los nombres de las clases abstractas aparecen en cursiva para distinguirlas de las clases concretas. Un diagrama puede incluir el pseudocódigo de la implementación de una operación; si es así, el código aparecerá en una caja con la esquina doblada unida por una línea discontinua a la operación que implementa.



Una clase mezclable [3] es aquélla pensada para proporcionar una interfaz o funcionalidad opcional a otras clases. Se parece a una clase abstracta en que no está pensada para que se creen instancias de ella. Las clases mezclables requieren herencia múltiple:



Herencia de clases frente a herencia de interfaces

Es importante entender la diferencia entre la *clase* de un objeto y su *tipo*.

La clase de un objeto define cómo se implementa un objeto. La clase define el estado interno del objeto y la implementación de sus operaciones. Por el contrario, el tipo de un objeto sólo se refiere a su interfaz —el conjunto de peticiones a las cuales puede responder —. Un objeto puede tener muchos tipos, y objetos de clases diferentes pueden tener el mismo tipo.

Por supuesto, hay una estrecha relación entre clase y tipo. Puesto que una clase define las operaciones que puede realizar un objeto, también define el tipo del objeto. Cuando decimos que un objeto es una instancia de una clase, queremos decir que el objeto admite la interfaz definida por la clase.

Lenguajes como C++ y Eiffel usan clases para especificar tanto el tipo de un objeto como su implementación. Los programadores de Smalltalk no declaran los tipos de las variables; por tanto, el compilador no comprueba que los tipos de los objetos asignados a una variable sean subtipos del tipo de la variable. Enviar un mensaje requiere comprobar que la clase del receptor implementa el mensaje, pero no que el receptor sea una instancia de una clase determinada.

También es importante comprender la diferencia entre la herencia de clases y la de interfaces (o subtipado). La herencia de clases define la implementación de un objeto en términos de la implementación de otro objeto. En resumen, es un mecanismo para compartir código y representación. Por el contrario, la herencia de interfaces (o subtipado) describe cuándo se puede usar un objeto en el lugar de otro.

Es fácil confundir estos dos conceptos, porque muchos lenguajes no hacen esta distinción explícita. En lenguajes como C++ y Eiffel, herencia significa tanto herencia de interfaces como de implementación. La manera normal de heredar de una interfaz en C++ es heredar públicamente de una clase que tiene funciones miembro virtuales (puras). La herencia de interfaces pura se puede simular en C++ heredando públicamente de clases abstractas puras. La herencia de implementación o de clases pura puede

simularse con herencia privada. En Smalltalk, la herencia significa simplemente herencia de implementación. Se pueden asignar instancias de cualquier clase a una variable siempre que sus instancias permitan las operaciones realizadas sobre el valor de la variable.

Si bien la mayoría de los lenguajes de programación no admiten la distinción entre herencia de interfaces y de implementación, la gente hace esa distinción en la práctica. Los programadores de Smalltalk normalmente tratan a las subclases como si fueran subtipos (aunque hay algunas conocidas excepciones [Coo92]); los programadores de C++ manipulan objetos a través de tipos definidos por clases abstractas.

Muchos de los patrones de diseño dependen de esta distinción. Por ejemplo, los objetos de una cadena en el patrón Chain of Responsibility (233) deben tener un tipo común, pero normalmente no comparten la misma implementación. En el patrón Composite (151), el Componente define una interfaz común, mientras que el Compuesto suele definir una implementación común. Los patrones Command (215). Observer (269), State (279), y Strategy (289) suelen implementarse con clases abstractas que son interfaces puras.

Programar para interfaces, no para una implementación

La herencia de clases no es mis que un mecanismo para extender la funcionalidad de una aplicación reutilizando la funcionalidad de las clases padres. Permite definir rápidamente un nuevo tipo de objetos basándose en otro, y obtener así nuevas implementaciones casi sin esfuerzo, al heredar la mayoría de lo que se necesita de clases ya existentes.

En cualquier caso, reutilizar la implementación es sólo la mitad de la historia. También es importante la capacidad de la herencia para definir familias de objetos con interfaces *idénticas* (normalmente heredando de una clase abstracta), al ser justamente en lo que se basa el polimorfismo.

Cuando la herencia se usa con cuidado (algunos dirían que cuando se usa *correctamente*), todas las clases que derivan de una clase abstracta compartirán su interfaz. Esto implica que una subclase simplemente añade o redefine operaciones y no oculta

operaciones de la clase padre. *Todas* las subclases pueden entonces responder a las peticiones en la interfaz de su clase abstracta, convirtiéndose así todas ellas en subtipos de la clase abstracta.

Manipular los objetos solamente en términos de la interfaz definida por las clases abstractas tiene dos ventajas:

1. Los clientes no tienen que conocer los tipos específicos de los objetos que usan, basta con que éstos se adhieran a la interfaz que esperan los clientes.

2. Los clientes desconocen las clases que implementan dichos objetos; sólo conocen las clases abstractas que definen la interfaz.

Esto reduce de tal manera las dependencias de implementación entre subsistemas que lleva al siguiente principio del diseño orientado a objetos reutilizable:

> *Programe para una interfaz, no para una implementación.*

Es decir, no se deben declarar las variables como instancias de clases concretas. En vez de eso, se ajustarán simplemente a la interfaz definida por una clase abstracta. Esto será algo recurrente en los patrones de diseño de este libro.

No obstante, es evidente que en algún lugar del sistema habrá que crear instancias de clases concretas (esto es, especificar una determinada implementación), y los patrones de creación (Abstract Factory (79), Builder (89), Factory Method (99), Prototype (109) y Singleton (119)) se encargan de eso. Al abstraer el proceso de creación de objetos, estos patrones ofrecen diferentes modos de asociar una interfaz con su implementación de manera transparente. Los patrones de creación aseguran que el sistema se escriba en términos de interfaces, no de implementaciones.

PONER A FUNCIONAR LOS MECANISMOS DE REUTILIZACIÓN

La mayoría de la gente comprende conceptos como objetos, interfaces, clases y herencia. La dificultad radica en aplicarlos para construir software flexible y reutilizable, y los patrones de diseño

pueden mostrar cómo hacerlo.

Herencia frente a Composición

Las dos técnicas más comunes para reutilizar funcionalidad en sistemas orientados a objetos son la herencia de clases y la **composición de objetos**. Como ya hemos explicado, la herencia de clases permite definir una implementación en términos de otra. A esta forma de reutilización mediante herencia se la denomina frecuentemente **reutilización de caja blanca**. El término "caja blanca" se refiere a la visibilidad: con la herencia, las interioridades de las clases padres suelen hacerse visibles a las subclases.

La composición de objetos es una alternativa a la herencia de clases. Ahora, la nueva funcionalidad se obtiene ensamblando o *componiendo* objetos para obtener funcionalidad más compleja. La composición de objetos requiere que los objetos a componer tengan interfaces bien definidas. Este estilo de reutilización se denomina reutilización de caja negra, porque los detalles internos de los objetos no son visibles. Los objetos aparecen sólo como "cajas negras".

Tanto la herencia como la composición tienen sus ventajas e inconvenientes. La herencia de clases se define estáticamente en tiempo de compilación y es sencilla de usar, al estar permitida directamente por el lenguaje de programación. La herencia de clases también hace que sea más fácil modificar la implementación que está siendo reutilizada. Cuando una subclase redefine alguna de las operaciones, puede afectar también a las operaciones de las que hereda, suponiendo que éstas llamen a alguna de las operaciones redefinidas.

Pero la herencia de clases también tiene inconvenientes. En primer lugar, no se pueden cambiar las implementaciones heredadas de las clases padre en tiempo de ejecución, porque la herencia se define en tiempo de compilación. En segundo lugar, y lo que generalmente es peor, las clases padre suelen definir al menos parte de la representación física de sus subclases. Como la herencia expone a una subclase los detalles de la implementación de su padre, suele decirse que "la herencia rompe la encapsulación" [Sny86]. La implementación de una subclase se liga de tal forma a

la implementación de su clase padre que cualquier cambio en la implementación del padre obligará a cambiar la subclase.

Las dependencias de implementación pueden causar problemas al tratar de reutilizar una subclase. Cuando cualquier aspecto de la implementación heredada no sea apropiado para nuevos dominios de problemas, la clase padre deberá ser escrita de nuevo o reemplazada por otra más adecuada. Esta dependencia limita la flexibilidad y la reutilización. Una solución a esto es heredar sólo de clases abstractas, ya que éstas normalmente tienen poca o ninguna implementación.

La composición de objetos se define dinámicamente en tiempo de ejecución a través de objetos que tienen referencias a otros objetos. La composición requiere que los objetos tengan en cuenta las interfaces de los otros, lo que a su vez requiere interfaces cuidadosamente diseñadas que no impidan que un objeto sea utilizado por otros. Pero hay una ventaja en esto: puesto que a los objetos se accede sólo a través de sus interfaces no se rompe su encapsulación. Cualquier objeto puede ser reemplazado en tiempo de ejecución por otro siempre que sean del mismo tipo. Además, como la implementación de un objeto se escribirá en términos de interfaces de objetos, las dependencias de implementación son notablemente menores.

La composición de objetos produce otro efecto en el diseño del sistema. Optar por la composición de objetos frente a la herencia de clases ayuda a mantener cada clase encapsulada y centrada en una sola tarea. De esta manera, nuestras clases y jerarquías de clases permanecerán pequeñas y será menos probable que se conviertan en monstruos inmanejables. Por otro lado, un diseño basado en la composición de objetos tendrá más objetos (al tener menos clases), y el comportamiento del sistema dependerá de sus relaciones en vez de estar definido en una clase.

Esto nos lleva a nuestro segundo principio del diseño orientado a objetos:

> *Favorecer la composición de objetos frente a la herencia de clases.*

Idealmente, sólo crearíamos nuevos componentes para lograr la reutilización. Deberíamos ser capaces de conseguir toda la

funcionalidad que necesitásemos simplemente ensamblando componentes existentes a través de la composición de objetos. Sin embargo, rara vez es éste el caso, puesto que el conjunto de componentes disponibles nunca es, en la práctica, lo suficientemente rico. Reutilizar mediante la herencia hace más fácil construir nuevos componentes que puedan ser combinados con los antiguos. La herencia y la composición trabajan por lo tanto juntas.

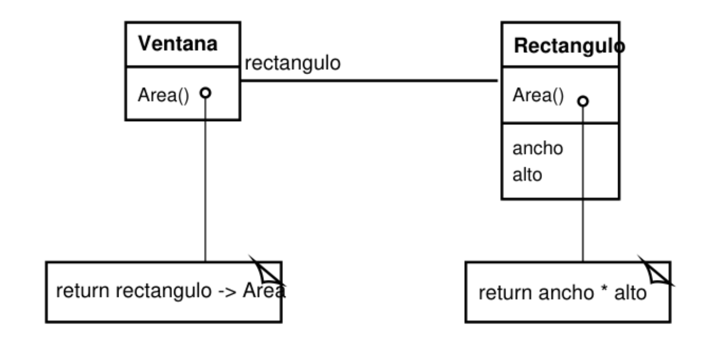
No obstante, nuestra experiencia es que los diseñadores abusan de la herencia como técnica de reutilización, y que los diseños suelen ser más reutilizables (y más simples) si dependen más de la composición de objetos. En los patrones de diseño se verá la composición de objetos aplicada una y otra vez.

Delegación

La **delegación** es un modo de lograr que la composición sea tan potente para la reutilización como lo es la herencia [Lie86, JZ9I]. Con la delegación, *dos* son los objetos encargados de tratar una petición: un objeto receptor delega operaciones en su **delegado**. Esto es parecido a la forma en que las subclases envían peticiones a las clases padres. Pero, con la herencia, una operación heredada siempre se puede referir al propio objeto a través de las variables miembro this de C++ o self de Smalltalk. Para lograr el mismo efecto con la delegación, el receptor se pasa a sí mismo al delegado, para que la operación delegada pueda referirse a él.

Por ejemplo, en vez de hacer que la clase Ventana sea una subclase de Rectángulo (porque resulta que las ventanas son rectangulares), la clase Ventana puede reutilizar el comportamiento de Rectángulo guardando una instancia de ésta en una variable y delegando en ella el comportamiento específico de los rectángulos. En otras palabras, en vez de hacer que una Ventana sea un Rectángulo, la Ventana contendrá un Rectángulo. Ahora Ventana debe reenviar las peticiones a su instancia de Rectángulo explícitamente, mientras que antes habría heredado esas operaciones.

El siguiente diagrama muestra la clase Ventana delegando su operación Area a una instancia de Rectángulo.



Una flecha lisa indica que una clase tiene una referencia a una instancia de otra clase. La referencia tiene un nombre opcional, en este caso “rectángulo”.

La principal ventaja de la delegación es que hace que sea fácil combinar comportamientos en tiempo de ejecución y cambiar la manera en que éstos se combinan. Nuestra ventana puede hacerse circular en tiempo de ejecución simplemente cambiando su instancia de Rectángulo por una instancia de Circulo, suponiendo que Rectángulo y Circulo tengan el mismo tipo.

La delegación tiene un inconveniente común a otras técnicas que hacen al software más flexible mediante la composición de objetos: el software dinámico y altamente parametrizado es más difícil de entender que el estático. Hay también ineficiencias en tiempo de ejecución, aunque las ineficiencias humanas son más importantes a largo plazo. La delegación es una buena elección de diseño sólo cuando simplifica más de lo que complica. No es fácil dar reglas que digan exactamente cuándo hay que usar delegación, porque su efectividad dependerá del contexto y de lo acostumbrados que estemos a ella. La delegación funciona mejor cuando se usa de manera muy estilizada, es decir, en patrones estándar.

Varios patrones de diseño usan la delegación. Los patrones State (279), Strategy (289) y Visitor (305) se basan en ella. En el patrón State, un objeto delega peticiones en un objeto Estado que representa su estado actual. En el patrón Strategy, un objeto delega una petición en un objeto que representa una estrategia para llevarla a cabo. Un objeto sólo tendrá un estado, pero puede tener muchas estrategias para diferentes peticiones. El propósito de ambos patrones es cambiar el comportamiento de un objeto cambiando los objetos en los que éste delega. En el patrón Visitor,

la operación que se realiza sobre cada elemento de una estructura de objetos siempre se delega en el objeto Visitante.

Otros patrones usan la delegación de manera menos notoria. El patrón Mediator (251) introduce un objeto que hace de mediador en la comunicación entre otros objetos. A veces, el objeto Mediador implementa operaciones simplemente redirigiéndolas a otros objetos; otras veces pasa una referencia a sí mismo, usando así verdadera delegación. El patrón Chain of Responsibility (205) procesa peticiones reenviándolas de un objeto a otro a través de una cadena. A veces la petición lleva consigo una referencia al objeto que recibió originalmente la petición, en cuyo caso el patrón está usando delegación. El patrón Bridge (141) desacopla una abstracción de su implementación. En el caso de que la abstracción y una implementación concreta estén muy relacionadas, la abstracción puede simplemente delegar operaciones en dicha implementación.

La delegación es un ejemplo extremo de composición de objetos. Muestra cómo siempre se puede sustituir la herencia por la composición de objetos como mecanismo de reutilización de código.

Herencia frente a Tipos Parametrizados

Otra técnica (no estrictamente orientada a objetos) para reutilizar funcionalidad es a través de los **tipos parametrizados**, también conocidos como **genéricos** (Ada, Eiffel) y **plantillas**[4] (C++). Esta técnica permite definir un tipo sin especificar todos los otros tipos que usa. Los tipos sin especificar se proporcionan como *parámetros* cuando se va a usar el tipo parametrizado. Por ejemplo, una clase Lista puede estar parametrizada por el tipo de los elementos que contiene. Para declarar una lista de enteros, proporcionaríamos el tipo “integer” como parámetro del tipo parametrizado Lista.

Para declarar una lista de objetos String, proporcionaríamos el tipo “String” como parámetro. El lenguaje de implementación creará una versión particularizada de la plantilla de la clase Lista para cada tipo de elemento.

Los tipos parametrizados nos dan una tercera forma (además de

la herencia de clases y la composición de objetos) de combinar comportamiento en sistemas orientados a objetos. Muchos diseños se pueden implementar usando alguna de estas tres técnicas. Para parametrizar una rutina de ordenación según el tipo de operación que usa para comparar elementos, podríamos hacer que la comparación fuese

1. una operación implementada por las subclases (una aplicación del patrón Template Method (299)),
2. responsabilidad de un objeto pasado a la rutina de ordenación (Strategy (289)), o
3. un argumento de una plantilla C++ o de un genérico de Ada que especifica el nombre de la función a llamar para comparar los elementos.

Hay diferencias importantes entre estas técnicas. La composición de objetos nos permite cambiar el comportamiento en tiempo de ejecución, pero también requiere indirección y puede ser menos eficiente. La herencia nos deja proporcionar implementaciones de operaciones predeterminadas y que las subclases las redefinan. Los tipos parametrizados permiten cambiar los tipos que puede utilizar una clase. Qué enfoque es mejor depende de nuestras restricciones de diseño e implementación.

Ninguno de los patrones de este libro trata de tipos parametrizados, aunque los usamos en ocasiones para personalizar la implementación C++ de algún patrón. Los tipos parametrizados no son necesarios en lenguajes como Smalltalk, que no tienen comprobación de tipos en tiempo de compilación.

ESTRUCTURAS QUE RELACIONAN TIEMPO DE EJECUCIÓN Y TIEMPO DE COMPILACIÓN

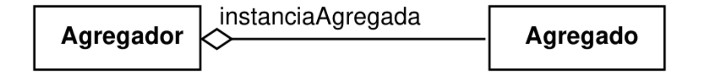
La estructura en tiempo de ejecución de un programa orientado a objetos suele guardar poco parecido con la estructura de su código. La estructura del código se fija en tiempo de compilación, y consiste en clases con relaciones de herencia estáticas. La estructura de tiempo de ejecución de un programa consiste en redes cambiantes de objetos que se comunican entre sí. De hecho, ambas estructuras son en gran medida independientes. Tratar de entender una a partir

de la otra es como tratar de entender el dinamismo de los ecosistemas vivos a partir de la taxonomía estática de plantas y animales, y viceversa.

Consideremos la distinción entre **agregación** y **asociación** [5] de objetos, y cómo se manifiestan esas diferencias en tiempo de ejecución y en tiempo de compilación. La agregación implica que un objeto posee a otro o que es responsable de él. Normalmente decimos que un objeto *tiene* a otro o que un objeto *es parte de* otro. La agregación implica que el objeto agregado y su propietario tienen la misma vida.

La asociación implica que un objeto simplemente *conoce a* otro. A veces, a la asociación también se la denomina relación de "uso". Los objetos así relacionados pueden pedirse operaciones entre sí, pero no son responsables unos de otros. Es una relación más débil que la agregación y representa mucho menor acoplamiento entre objetos.

En nuestros diagramas, una flecha con la línea continua denota asociación, mientras que una flecha con un rombo en su base denota agregación:



Es fácil confundir agregación y asociación, ya que muchas veces se implementan de la misma forma. En Smalltalk, todas las variables son referencias a otros objetos, no hay distinción en el lengua je de programación entre agregación y asociación. En C++, la agregación se puede implementar definiendo variables miembro que sean verdaderas instancias, pero es más frecuente definirlas como punteros o referencias a instancias. La asociación se implementa también con punteros y referencias.

En última instancia, la asociación y la agregación quedan determinadas más por su intención que por mecanismos explícitos del lenguaje. La distinción puede ser difícil de ver en la estructura de tiempo de compilación, pero es notable. Las relaciones de agregación tienden a ser menos y más permanentes que las de asociación. Las asociaciones, por el contrario, se hacen y deshacen mucho más frecuentemente, y algunas veces sólo existen mientras

dura una operación. También son más dinámicas, haciéndolas más difíciles de distinguir en el código fuente.

Con esa disparidad entre las estructuras de tiempo de ejecución y compilación de un programa, es evidente que el código no lo revelará todo acerca de cómo funciona un sistema. La estructura de tiempo de ejecución de un sistema debe ser impuesta más por el diseñador que por el lenguaje. Las relaciones entre objetos y sus tipos deben ser diseñadas con mucho cuidado, porque van a determinar la calidad de la estructura de tiempo de ejecución.

Muchos patrones de diseño (en concreto aquellos que tienen ámbito de objeto) representan explícitamente la distinción entre ambas estructuras. El patrón Composite (151) y el Decorator (161) son especialmente útiles para construir estructuras de tiempo de ejecución complejas. El Observer (269) involucra estructuras de tiempo de ejecución que suelen ser difíciles de entender si no se conoce el patrón. El patrón Chain of Responsibility (205) también produce patrones de comunicación que la herencia no pone de manifiesto. En general, las estructuras de tiempo de ejecución no están claras en el código hasta que se comprenden los patrones.

DISEÑAR PARA EL CAMBIO

La clave para maximizar la reutilización reside en anticipar nuevos requisitos y cambios en los requisitos existentes, y en diseñar los sistemas de manera que puedan evolucionar en consecuencia.

Para diseñar un sistema que sea robusto a dichos cambios hay que tener en cuenta cómo puede necesitar cambiar el sistema a lo largo de su vida. Un diseño que no tenga en cuenta el cambio sufre el riesgo de tener que ser rediseñado por completo en el futuro. Dichos cambios pueden involucrar redefiniciones y reimplementaciones de clases, modificar los clientes y volver a hacer pruebas. El rediseño afecta a muchas partes del sistema software, por lo que los cambios no previstos siempre resultan costosos.

Los patrones de diseño ayudan a evitar esto al asegurar que un sistema pueda cambiar de formas concretas. Cada patrón de diseño deja que algún aspecto de la estructura del sistema varíe independientemente de los otros, haciendo así al sistema más robusto frente a un tipo de cambio concreto.

A continuación se presentan algunas las causas comunes de rediseño junto con los patrones de diseño que lo resuelven:

1. *Crear un objeto especificando su clase explícitamente*. Especificar un nombre de clase al crear un objeto nos liga a una implementación concreta en vez de a una interfaz. Esto puede complicar los cambios futuros. Para evitarlo, debemos crear los objetos indirectamente.

   Patrones de diseño: Abstract Factory (79), Factory Method (99), Prototype (109).

2. *Dependencia de operaciones concretas*. Cuando especificamos una determinada operación, estamos ligándonos a una forma de satisfacer una petición. Evitando ligar las peticiones al código, hacemos más fácil cambiar el modo de satisfacer una petición, tanto en tiempo de compilación como en tiempo de ejecución.

   Patrones de diseño: Chain of Responsibility (205), Command (215).

3. *Dependencia de plataformas hardware o software*. Las interfaces externas de los sistemas operativos y las interfaces de programación de aplicaciones (API) varían para las diferentes plataformas hardware y software. El software que depende de una plataforma concreta será más difícil de portar a otras plataformas. Incluso puede resultar difícil mantenerlo actualizado en su plataforma nativa. Por tanto, es importante diseñar nuestros sistemas de manera que se limiten sus dependencias de plataforma.

   Patrones de diseño: Abstract Factory (79), Bridge (141).

4. *Dependencia de las representaciones o implementaciones de objetos*. Los clientes de un objeto que saben cómo se representa, se almacena, se localiza o se implementa, quizá deban ser modificados cuando cambie dicho objeto. Ocultar esta información a los clientes previene los cambios en cascada.

Patrones de diseño: Abstract Factory (79), Bridge (141), Memento (261), Proxy (191).

5. *Dependencias algorítmicas*. Muchas veces los algoritmos se amplían, optimizan o sustituyen por otros durante el desarrollo y posterior reutilización. Los objetos que dependen de un algoritmo tendrán que cambiar cuando éste cambie. Por tanto, aquellos algoritmos que es probable que cambien deberían estar aislados.

   Patrones de diseño: Builder (89), Iterator (237), Strategy (289), Template Method (299), Visitor (305).

6. *Fuerte acoplamiento*. Las clases que están fuertemente acopladas son difíciles de reutilizar por separado, puesto que dependen unas de otras. El fuerte acoplamiento lleva a sistemas monolíticos, en los que no se puede cambiar o quitar una clase sin entender y cambiar muchas otras. El sistema se convierte así en algo muy denso que resulta difícil de aprender, portar y mantener.

   El bajo acoplamiento aumenta la probabilidad de que una clase pueda ser reutilizada ella sola y de que un sistema pueda aprenderse, portarse, modificarse y extenderse más fácilmente. Los patrones de diseño hacen uso de técnicas como al acoplamiento abstracto y la estructuración en capas para promover sistemas escasamente acoplados.

   Patrones de Diseño: Abstract Factory (79), Bridge (141), Chain of Responsibility (205), Command (215), Facade (171), Mediator (251), Observer (269).

7. *Añadir funcionalidad mediante la herencia*. Particularizar un objeto derivando de otra clase no suele ser fácil. Cada nueva clase tiene un coste de implementación (inicialización, finalización, etc.). Definir una subclase también requiere un profundo conocimiento de la clase padre. Por ejemplo, redefinir una operación puede requerir redefinir otra, o tener que llamar a una operación heredada. Además, la herencia puede conducir a una explosión de clases, ya que una simple extensión puede

obligar a introducir un montón de clases nuevas.

La composición de objetos en general y la delegación en particular proporcionan alternativas flexibles a la herencia para combinar comportamiento. Se puede añadir nueva funcionalidad a una aplicación componiendo los objetos existentes de otra forma en vez de definir subclases nuevas de otras clases existentes. No obstante, también es cierto que un uso intensivo de la composición de objetos puede hacer que los diseños sean más difíciles de entender. Muchos patrones de diseño producen diseños en los que se puede introducir nueva funcionalidad simplemente definiendo una subclase y componiendo sus instancias con otras existentes. Patrones de diseño: Bridge (141), Chain of Responsibility (205), Composite (151), Decorator (161), Observer (269), Strategy (289).

8. *Incapacidad para modificar las clases convenientemente*. A veces hay que modificar una clase que no puede ser modificada convenientemente. Quizá necesitemos el código fuente y no lo tengamos (como puede ser el caso de una biblioteca de clases comercial). O tal vez cualquier cambio requeriría modificar muchas de las subclases existentes. Los patrones de diseño ofrecen formas de modificar las clases en tales circunstancias.

   Patrones de diseño: Adapter (131), Decorator (161), Visitor (305).

Estos ejemplos reflejan la flexibilidad que los patrones de diseño pueden ayudamos a conseguir en nuestro software. Cómo sea esta flexibilidad de crucial depende del tipo de software que estemos desarrollando. Echemos un vistazo al papel que desempeñan los patrones de diseño en el desarrollo de tres amplias clases de software: programas de aplicación, *toolkits y frameworks*.

Programas de aplicación

Si estamos construyendo un programa de aplicación, como un editor de documentos o una hoja de cálculo, la reutilización interna,

la facilidad de mantenimiento y la extensión son las principales prioridades. La reutilización interna hace que no haya que diseñar e implementar más de lo estrictamente necesario. Los patrones de diseño que reducen dependencias pueden aumentar la reutilización interna. Un acoplamiento más bajo aumenta la probabilidad de que una clase de objeto pueda cooperar con otras. Por ejemplo, cuando eliminamos las dependencias de operaciones específicas aislando y encapsulando cada operación, estamos haciendo que sea más fácil reutilizar una operación en diferentes contextos. Lo mismo ocurre cuando eliminamos dependencias algorítmicas y de representación.

Los patrones de diseño también hacen que una aplicación sea más fácil de mantener cuando se usan para limitar las dependencias de plataforma y para organizar un sistema en capas. Mejoran la extensibilidad al ilustrar cómo extender jerarquías de clases y cómo explotar la composición de objetos. Reducir el acoplamiento también mejora la extensibilidad. Extender una clase aislada es más fácil si ésta no depende de otras muchas clases.

Toolkits [6]

Muchas veces una aplicación incorpora clases de una o más bibliotecas de clases predefinidas llamadas ***toolkits***. Un toolkit es un conjunto de clases relacionadas y reutilizables diseñadas para proporcionar funcionalidad útil de propósito general. Un ejemplo de toolkit es un conjunto de clases para tratar con listas, tablas asociativas, pilas y similares. La biblioteca de flujos de entrada/salida de C++ es otro ejemplo. Los toolkits no imponen un diseño particular en una aplicación; simplemente proporcionan funcionalidad que puede ayudar a que la aplicación haga su trabajo. Nos permiten, como desarrolladores, evitar recodificar funcionalidad común. Los toolkits se centran en la *reutilización de código*, siendo el equivalente orientado a objetos de las bibliotecas de subrutinas.

Diseñar toolkits es posiblemente más difícil que diseñar una aplicación, ya que aquéllos tienen que funcionar en muchas aplicaciones para ser útiles. Además, el creador del toolkit no puede saber cuáles van a ser esas aplicaciones o cuáles serán sus necesidades especiales. Eso hace que lo más importante sea evitar

suposiciones y dependencias que puedan limitar la flexibilidad del toolkit y consecuentemente su aplicabilidad y efectividad.

Frameworks [7]

Un ***framework*** es un conjunto de clases cooperantes que constituyen un diseño reutilizable para una clase específica de software [Deu89, JF88]. Por ejemplo, un framework puede estar orientado a la construcción de editores gráficos para dominios diferentes, como el dibujo artístico, la composición musical y el CAD [VL90, Joh92]. Otro puede ayudar a construir compiladores para diferentes lenguajes de programación y máquinas de destino [JML92], Otro podría ayudar a construir aplicaciones de modelado financiero [BE93]. Personalizamos un framework para una aplicación concreta creando subclases específicas de la aplicación de clases abstractas del framework.

El framework determina la arquitectura de nuestra aplicación. Definirá la estructura general, su partición en clases y objetos, las responsabilidades clave, cómo colaboran las clases y objetos y el hilo de control. Un framework predefine estos parámetros de diseño de manera que el diseñador o el programador de la aplicación puedan concentrarse en las particularidades de dicha aplicación. El framework representa las decisiones de diseño que son comunes a su dominio de aplicación. Los frameworks hacen hincapié así en la reutilización del diseño frente a la reutilización de código, si bien un framework incluirá normalmente subclases concretas listas para trabajar con ellas inmediatamente

La reutilización a este nivel lleva a una inversión de control entre la aplicación y el software en el que se basa. Cuando utilizamos un toolkit (o una biblioteca de subrutinas tradicional) escribimos el cuerpo principal de la aplicación y llamamos al código que queremos reutilizar. Cuando usamos un framework podemos reutilizar el cuerpo principal y escribir el código al que llama. Habrá que escribir operaciones con nombres específicos y convenios de llamada, pero eso reduce las decisiones de diseño que hay que tomar.

Como resultado, no sólo se pueden construir aplicaciones más rápidamente, sino que las aplicaciones tienen estructuras parecidas, por lo que son más fáciles de mantener y resultan más consistentes

para los usuarios. Por otro lado, perdemos algo de libertad creativa, puesto que muchas decisiones de diseño ya han sido tomadas por nosotros.

Si las aplicaciones son difíciles de diseñar, y los toolkilts más difíciles todavía, los frameworks son los más difíciles de todos. Un diseñador de frameworks supone que una misma arquitectura servirá para todas las aplicaciones de ese dominio. Cualquier cambio sustantivo en el diseño del framework reduciría considerablemente sus beneficios, puesto que su principal contribución a una aplicación es la arquitectura que define. Por tanto, es necesario diseñar el framework para que sea tan flexible y extensible como sea posible.

Además, como las aplicaciones dependen tanto del framework, son particularmente sensibles a los cambios en las interfaces de éste. A medida que un framework evoluciona, las aplicaciones tienen que evolucionar con él. Esto hace que un bajo acoplamiento sea lo más importante de todo; si no, hasta el más mínimo cambio del framework tendrá importantes repercusiones.

Los problemas de diseño que acabamos de estudiar son fundamentales en el diseño del framework. Un framework que los resuelva usando patrones de diseño es mucho más probable que consiga un alto grado de reutilización del diseño y del código que otro que no lo haga. Los frameworks maduros normalmente incorporan varios patrones de diseño. Los patrones ayudan a hacer que la arquitectura del framework sirva para muchas aplicaciones diferentes sin necesidad de rediseño.

Una ventaja añadida se produce cuando el framework se documenta con los patrones de diseño que usa [BJ94], Quien conoce los patrones aprende cómo está hecho el framework mucho más rápidamente. Incluso quienes no conocen los patrones se pueden beneficiar de la estructura que éstos confieren a la documentación del framework. Mejorar la documentación es importante para cualquier tipo de software, pero es particularmente importante en el caso de los frameworks, ya que suelen tener una curva de aprendizaje que es necesario superar para que comiencen a ser útiles. Si bien los patrones de diseño puede que no allanen del todo dicha curva de aprendizaje, sí la pueden hacer menos pendiente al hacer explícitos los elementos clave del diseño del

framework.

Como los patrones y los frameworks tienen similitudes, mucha gente se pregunta en qué se diferencian, si es que se diferencian en algo. Son diferentes en tres aspectos fundamentales:

1. *Los patrones de diseño son más abstractos que los frameworks*. Los frameworks pueden plasmarse en código, pero sólo los ejemplos de los patrones pueden ser plasmados en código. Uno de los puntos fuertes de los frameworks es que se pueden escribir en lenguajes de programación y de ese modo ser no sólo estudiados, sino ejecutados y reutilizados directamente. Por el contrarío, los patrones de diseño de este libro tienen que ser implementados cada vez que se emplean. Los patrones de diseño también reflejan la intención, las ventajas e inconvenientes y las consecuencias de un diseño.
2. *Los patrones de diseño son elementos arquitectónicos más pequeños que los frameworks*. Un framework típico contiene varios patrones de diseño, pero lo contrarío nunca es cierto.
3. *Los patrones de diseño están menos especializados que los frameworks*. Los frameworks siempre tienen un dominio de aplicación concreto. Un editor gráfico se puede usaren una simulación de una fábrica, pero nadie lo tomará por un framework de simulación, Sin embargo, los patrones de diseño de este catálogo se pueden usar en prácticamente cualquier tipo de aplicación. Aunque es posible tener patrones de diseño más especializados que los nuestros (como, por ejemplo, patrones de diseño para programación concurrente), incluso éstos no imponen una arquitectura de aplicación como haría un framework.

Los frameworks se están convirtiendo en algo cada vez más común e importante. Son el modo en que los sistemas orientados a objetos consiguen la mayor reutilización. Las aplicaciones orientadas a objetos más grandes terminarán consistiendo en capas de frameworks que cooperan unos con otros. La mayoría del diseño y del código de una aplicación vendrá dado o estará influido por los frameworks que utilice.

1.7. CÓMO SELECCIONAR UN PATRÓN DE DISEÑO

Con más de 20 patrones de diseño para elegir en el catálogo, puede ser difícil encontrar aquél que resuelva un problema de diseño concreto, especialmente si el catálogo es nuevo y desconocido para el lector. A continuación se muestran diferentes enfoques para encontrar el patrón de diseño que se adecúe a su problema:

- *Considere cómo los patrones de diseño solucionan problemas de diseño*. En la Sección 1.6 se vio cómo los patrones de diseño ayudan a encontrar los objetos apropiados, a determinar la granularidad, a especificar interfaces de objetos y otros aspectos en los que los patrones ayudan a resolver problemas de diseño. Consultar estas discusiones puede ayudarle a guiar su búsqueda del patrón adecuado.
- *Hojee las secciones Propósito*. La Sección 1.4 (página 7) enumera las secciones Propósito de todos los patrones del catálogo. Lea el propósito de cada patrón para encontrar uno o varios que parezcan relevantes para su problema. Puede usar el esquema de clasificación de la Tabla 1.1 (página 9) para guiar su búsqueda.
- *Estudie cómo se interrelacionan los patrones*. La Figura 1.1 (página 11) muestra gráficamente las relaciones entre los patrones de diseño. Estudiar dichas relaciones puede ayudarle a dirigirse al patrón o grupo de patrones apropiado.
- *Estudie patrones de propósito similar*. El catálogo (página 71) tiene tres capítulos: uno para patrones de creación, otro para los patrones estructurales y un tercero para patrones de comportamiento. Cada capítulo comienza con unos comentarios de introducción sobre los patrones y termina con una sección que los compara y contrasta. Estas secciones le ayudan a comprender las similitudes y diferencias entre patrones de propósito similar.
- *Examine una causa de rediseño*. Observe las causas de rediseño que comienzan en la página 21 para ver si su problema involucra a una o varias de ellas. Una vez hecho

eso, vea los patrones que ayudan a evitar las causas de rediseño.

- *Piense qué debería ser variable en su diseño*. Este enfoque es el opuesto a centrarse en las causas de rediseño. En vez de tener en cuenta qué podría *forzar* un cambio en el diseño, piense en qué quiere que *pueda* ser cambiado sin necesidad de rediseñar. Se trata de centrarse en encapsular el concepto que puede variar, un tema común a muchos patrones de diseño. La Tabla 1.2 enumera los aspectos de diseño que los patrones permiten variar de forma independiente, es decir, sin rediseño.

Tabla 1.2: Aspectos de diseño que los patrones de diseño permiten modificar

Propósito	**Patrones de diseño**	**Aspectos que pueden variar**
De creación	Abstract Factory (79)	la familia de los objetos producidos
	Builder (89)	cómo se crea un objeto compuesto
	Factory Method (99)	la subclase del objeto que es instanciado
	Prototype (109)	la clase del objeto que es instanciado
	Singleton (119)	la única instancia de una clase
Estructurales	Adapter (131)	la interfaz de un objeto
	Bridge (141)	la implementación de un objeto
	Composite (151)	la estructura y composición de un objeto
	Decorator (161)	las responsabilidades de un objeto sin usar la herencia
	Facade (171)	la interfaz de un subsistema
	Flyweight (179)	el coste de almacenamiento de los objetos
	Proxy (191)	cómo se accede a un objeto; su ubicación
De comportamiento	Chain of Responsibility (223)	el objeto que puede satisfacer una petición
	Command (215)	cuándo y cómo se satisface una petición
	Interpreter (225)	la gramática e

	interpretación de un lenguaje
Iterator (237)	cómo se recorren los elementos de un agregado
Mediator (251)	qué objetos interactúan entre sí, y cómo
Memento (261)	qué información privada se almacena fuera de un objeto, y cuándo
Observer (269)	el número de objetos que dependen de otro; cómo se mantiene actualizado el objeto dependiente
State (279)	el estado de un objeto
Strategy (289)	un algoritmo
Template Method (299)	los pasos de un algoritmo
Visitor (305)	las operaciones que pueden aplicarse a objetos sin cambiar sus clases

1.8. CÓMO USAR UN PATRÓN DE DISEÑO

Una vez que haya elegido un patrón de diseño, ¿cómo usarlo? Lo que sigue a continuación es un enfoque paso a paso para aplicar un patrón de diseño de manera efectiva:

1. *Lea el patrón de principio a fin para tener una perspectiva*. Preste particular atención a las secciones de Aplicabilidad y Consecuencias para asegurarse de que el patrón es el adecuado para su problema.
2. *Vuelva atrás y estudie las secciones de Estructura, Participantes y Colaboraciones*. Asegúrese de entender las clases y objetos del patrón y cómo se relacionan entre ellos.
3. *Examine la sección Código de Ejemplo para ver un ejemplo concreto del patrón en código*. Estudiar el código ayuda a entender cómo implementar el patrón.
4. *Elija nombres significativos en el contexto de la aplicación para los participantes en el patrón*. Los nombres de los participantes de los patrones de diseño normalmente son

demasiado abstractos como para aparecer directamente en una aplicación. Sin embargo, es útil incorporar el nombre del participante en el nombre que aparece en la aplicación. Eso ayuda a hacer el patrón más explícito en la implementación. Por ejemplo, si usa el patrón Strategy para un algoritmo de composición de texto, es posible que tenga las clases EstrategiaComposicionSimple o EstrategiaComposicionTeX.

5. *Defina las clases*. Declare sus interfaces, establezca sus relaciones de herencia y defina las variables de instancia que representan datos y referencias de objetos. Identifique las clases existentes en su aplicación a las que afectará el patrón y modifíquelas en consecuencia.
6. *Defina nombres específicos de la aplicación para las operaciones del patrón*. Los nombres generalmente dependen de la aplicación. Use las responsabilidades y colaboraciones asociadas con cada operación como una guía. También debe ser coherente en sus convenciones de nombres. Por ejemplo, podría usar el prefijo “Crear-” de forma constante para denotar un método de fabricación.
7. *Implemente las operaciones para llevar a cabo las responsabilidades y colaboraciones del patrón*. La sección Implementación ofrece pistas que le guiarán en la implementación. También pueden serle de ayuda los ejemplos de la sección Código de Ejemplo.

Éstas son sólo unas directrices generales que le pueden servir para empezar. A medida que pase el tiempo desarrollará su propio método para trabajar con patrones de diseño.

Ninguna discusión sobre cómo usar patrones de diseño estaría completa sin unas pocas palabras sobre cómo *no* usarlos. Los patrones de diseño no deberían ser aplicados indiscriminadamente. Muchas veces éstos consiguen la flexibilidad y la variabilidad a costa de introducir niveles adicionales de indirección, y eso puede complicar un diseño o disminuir el rendimiento. Un patrón de diseño sólo debería ser aplicado cuando la flexibilidad que proporcione sea realmente necesaria. Las secciones de Consecuencias son las más valiosas a la hora de evaluar los beneficios y los costes de un patrón.

CAPÍTULO 2

Un caso de estudios: diseño de un editor de documentos

CONTENIDO DEL CAPÍTULO

Este capítulo presenta un caso de estudio consistente en el diseño de un editor de documentos *What-You-See-Is-What-You-Get*[8] (o "WYSIWYG") llamado Lexi[9]. Veremos cómo los patrones de diseño representan soluciones a problemas de diseño de Lexi y otras aplicaciones similares. Al final del capítulo habrá experimentado con ocho patrones, aprendiéndolos mediante ejemplos.

La Figura 2.1 muestra la interfaz de usuario de Lexi. Una representación WYSIWYG del documento ocupa el área rectangular grande del centro. El documento puede combinar texto y gráficos con diferentes estilos de formateado. Alrededor del documento están los típicos menús desplegables y barras de desplazamiento, más una serie de iconos de página que permiten ir a una determinada página del documento.

2.1. PROBLEMAS DE DISEÑO

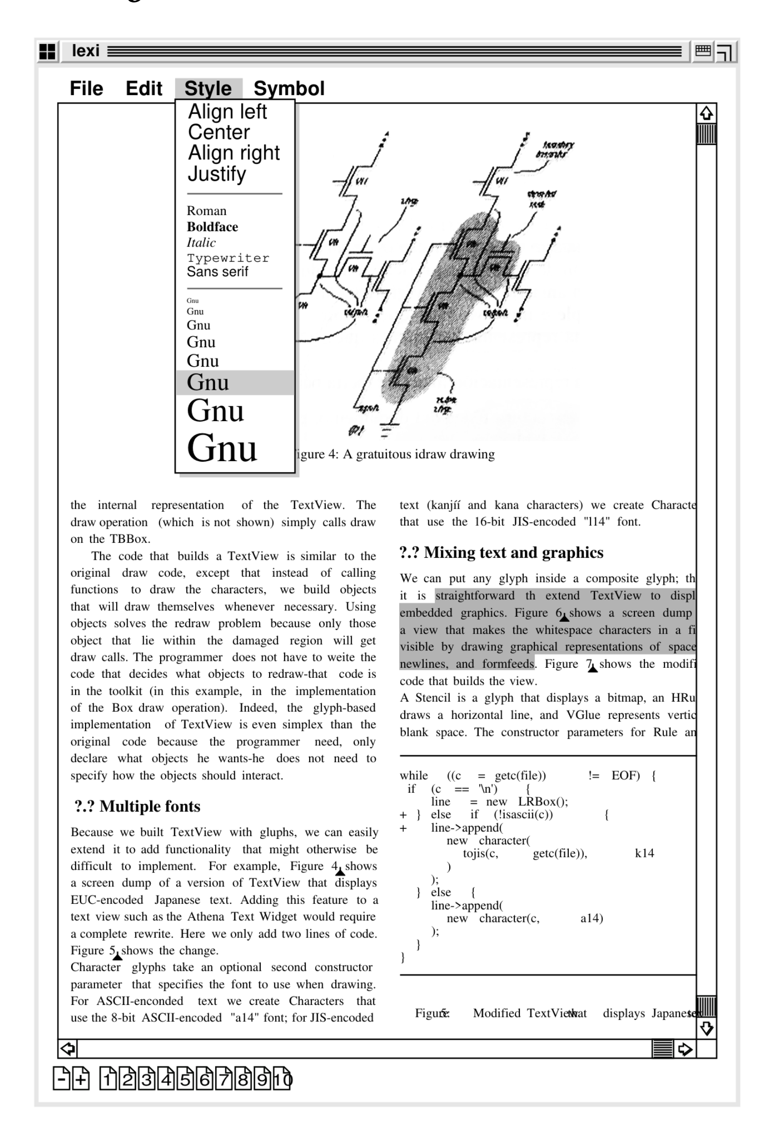
A continuación examinaremos siete problemas del diseño de Lexi:

1. *Estructura de los documentos*. La elección de la representación interna del documento afecta a prácticamente todos los aspectos del diseño de Lexi. Tanto la edición como el formateado, visualización y análisis del texto necesitarán recorrer dicha representación. La manera en que organicemos esta información influirá, por tanto, en el resto de la aplicación.
2. *Formateado*. ¿Cómo coloca realmente Lexi el texto y los gráficos en líneas y columnas? ¿Qué objetos son responsables de llevar a cabo las diferentes políticas de formateado? ¿Cómo interactúan éstas con la representación interna del documento?

3. *Adornos de la interfaz de usuario*. La interfaz de usuario de Lexi incluye barras de desplazamiento, bordes y sombras que mejoran la apariencia del documento WYSIWYG. Es probable que dichos elementos gráficos cambien a medida que evolucione la interfaz, de ahí la importancia de poderlos añadir y quitar fácilmente, sin que esto afecte al resto de la aplicación.
4. *Permitir múltiples estándares de interfaz de usuario* **[10]**. Lexi debería poder adaptarse fácilmente a diferentes estándares de interfaz de usuario, tales como Motif o Presentation Manager (PM), sin falta de grandes modificaciones.
5. *Permitir múltiples sistemas de ventanas*. Los distintos estándares de interfaz de usuario suelen estar implementados en sistemas de ventanas diferentes. El diseño de Lexi debería ser tan independiente como fuera posible del sistema de ventanas.
6. *Operaciones de usuario*. El usuario controla Lexi a través de varios elementos de la interfaz, incluyendo botones y menús desplegables. La funcionalidad que hay tras estos elementos de la interfaz se encuentra repartida entre los objetos de la aplicación. El objetivo es proporcionar un mecanismo uniforme tanto para acceder a esta funcionalidad dispersa como para deshacer sus efectos.
7. *Comprobación ortográfica y separación de palabras*. ¿Cómo permite Lexi operaciones de análisis tales como la comprobación de palabras mal escritas o determinar los puntos donde debe insertarse un guión de separación? ¿Cómo podemos minimizar el número de clases que deben ser modificadas para añadir una nueva operación de este tipo?

Discutiremos estos problemas de diseño en las secciones siguientes. Cada problema tiene asociados una serie de objetivos, además de restricciones sobre cómo lograr dichos objetivos. Los objetivos y sus restricciones serán explicados en detalle antes de proponer una solución concreta. El problema y su solución ilustrarán uno o más patrones de diseño. La discusión de cada problema culminará con una breve introducción a los patrones relevantes.

Figura 2.1: La interfaz de usuario de Lexi



2.2. ESTRUCTURA DEL DOCUMENTO

Un documento al final no es más que una disposición de elementos gráficos básicos, como caracteres, líneas, polígonos y otras formas.

Estos elementos representan toda la información contenida en el documento. Sin embargo, el autor del documento no los suele ver como elementos gráficos, sino en términos de la estructura física del documento —líneas, columnas, figuras, tablas y otras subestructuras —[11]. A su vez, estas subestructuras tienen otras subestructuras propias, y así sucesivamente.

La interfaz de usuario de Lexi debería permitir manipular estas subestructuras directamente. Por ejemplo, un usuario debería ser capaz de manipular un diagrama como una unidad, en vez de como una colección de primitivas gráficas individuales, o de poder referirse a una tabla como un todo y no como un amasijo de texto y gráficos sin estructura. Esto contribuye a que la interfaz de Lexi sea más simple e intuitiva. Para que la implementación tenga unas características similares debemos elegir una representación interna que se corresponda con la estructura física del documento.

En concreto, la representación interna debería permitir lo siguiente:

- Mantener la estructura física del documento, es decir, la disposición de texto y gráficas en líneas, columnas, tablas, etc.
- Generar y presentar visualmente el documento.
- Establecer una correspondencia entre las posiciones en pantalla y los elementos de la representación interna. De esta manera, Lexi puede determinar a qué se está refiriendo el usuario cuando apunta a algo en la representación visual.

Además de estos objetivos hay una serie de restricciones. En primer lugar, deberíamos tratar uniformemente al texto y a los gráficos. La interfaz de usuario permite añadir texto a los gráficos y viceversa, por lo que deberíamos evitar tratar los gráficos como un caso especial de texto, o el texto como un caso especial de gráfico. De no hacerlo así, acabaremos por tener mecanismos redundantes de formateado y manipulación. Debe ser suficiente con un único conjunto de mecanismos para el texto y los gráficos.

En segundo lugar, nuestra implementación no debería distinguir, en la representación interna, entre elementos individuales y grupos de elementos. Lexi tendría que ser capaz de tratar de manera uniforme elementos simples y compuestos, permitiendo así

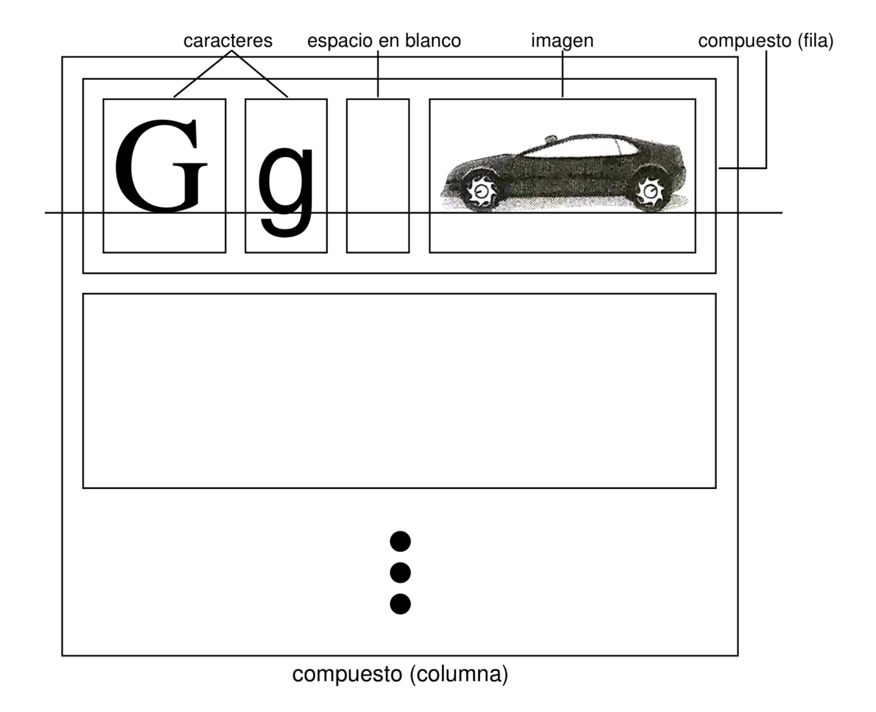
documentos todo lo complejos que se quiera. Por ejemplo, el décimo elemento de la línea cinco, columna dos, podría ser un carácter individual o un intrincado diagrama con muchos subelementos. En la medida en que sepamos que dicho elemento puede dibujarse a sí mismo y especificar sus dimensiones, su complejidad no influye en cómo y dónde debe aparecer en la página.

Sin embargo, opuesta a esta segunda restricción está la necesidad de analizar el texto para asuntos como los errores ortográficos y los puntos potenciales de división de palabras con guiones. Aunque normalmente no nos preocuparemos de si el elemento de una línea es un objeto simple o compuesto, a veces un análisis depende de los objetos que están siendo analizados. Tiene poco sentido, por ejemplo, comprobar la ortografía de un polígono, o dividirlo con un guión al final de la línea. El diseño de la representación interna debería tener en cuenta estas y otras restricciones, potencialmente contradictorias.

COMPOSICION RECURSIVA

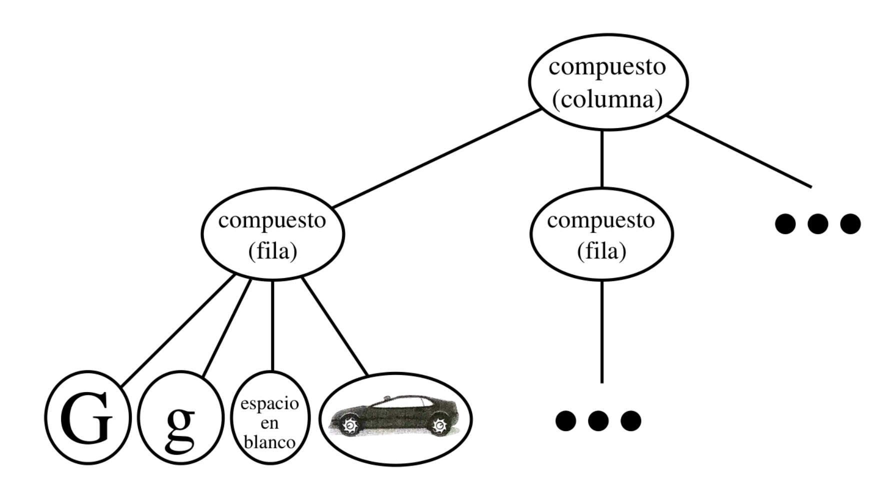
Una manera sencilla de representar jerárquicamente información estructurada es mediante una técnica denominada composición recursiva, que consiste en construir objetos cada vez más complejos a partir de otros más simples. La composición recursiva nos da la posibilidad de componer un documento a partir de elementos gráficos. Como primer paso, podemos disponer una serie de caracteres y gráficos de izquierda a derecha para formar una línea del documento. Después, podemos colocar varias líneas formando una columna, varias columnas formando una página, y así sucesivamente (*véase* la Figura 2.2).

Figura 2.2: Composición recursiva de texto y gráficos



Podemos representar esta estructura física dedicando un objeto para cada elemento importante. Eso incluye no sólo los elementos visibles como caracteres y gráficos, sino también los elementos estructurales, invisibles —líneas y columnas—. El resultado es la estructura de objetos mostrada en la Figura 2.3.

Figura 2.3: Estructura de objetos para la composición recursiva de texto y gráficos



Al utilizar un objeto para cada carácter y elemento gráfico del documento, estamos llevando la flexibilidad en el diseño de Lexi a su grado máximo. Podemos tratar al texto y a los gráficos de manera uniforme con respecto a cómo se dibujan, se formatean y se insertan unos en otros. Y podemos ampliar Lexi para que admita nuevos juegos de caracteres sin afectar al resto de la funcionalidad. La estructura de objetos de Lexi mimetiza la estructura del documento físico.

Este enfoque tiene dos repercusiones importantes. La primera es obvia: los objetos necesitan sus correspondientes clases. La segunda, que puede resultar menos evidente, es que estas clases deben tener interfaces compatibles, ya que queremos tratar uniformemente a los objetos. La manera de hacer que las interfaces sean compatibles en un lenguaje como C++ es relacionar las clases a través de la herencia.

GLIFOS [12]

Definiremos una clase abstracta Glifo para todos los objetos que pueden aparecer en la estructura de un documento [13]. Sus subclases definen tanto elementos gráficos primitivos (por ejemplo, caracteres e imágenes) como elementos estructurales (por ejemplo, filas y columnas). La Figura 2.4 muestra una parte representativa de la jerarquía de las clase Glifo, y la Tabla 2.1 presenta la interfaz básica de un glifo más detalladamente, empleando notación C++. [14]

Figura 2.4: Parte de la jerarquía de clases de Glifo

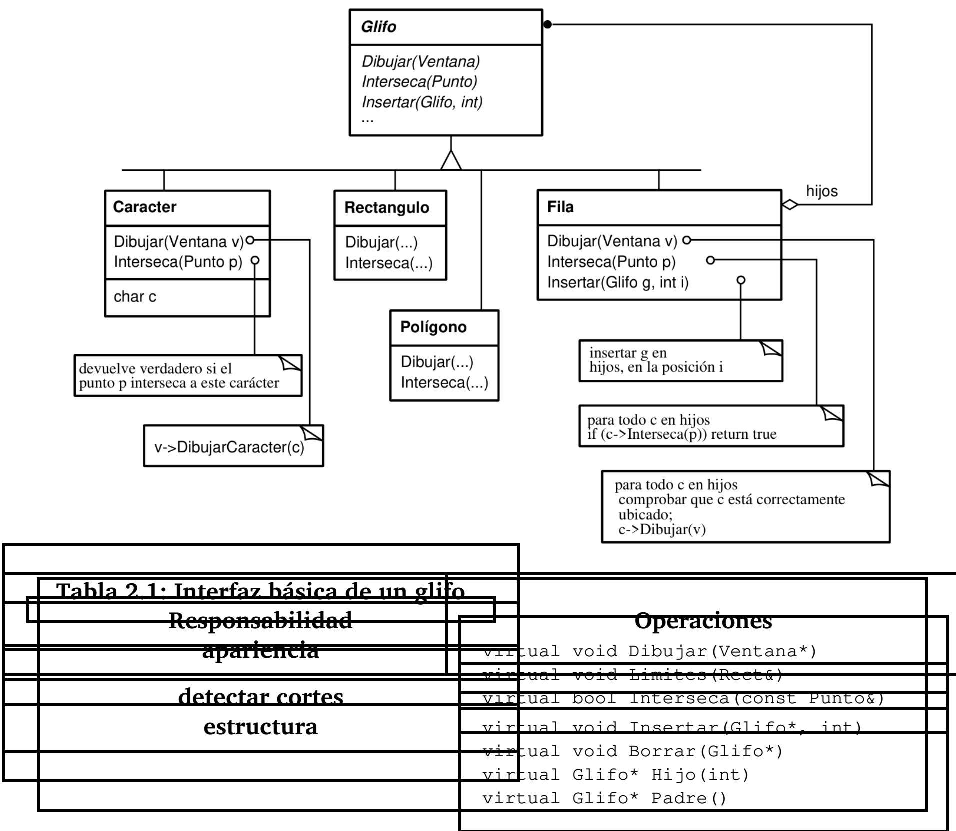


Tabla 2.1: Interfaz básica de un glifo

Responsabilidad	Operaciones
apariencia	`virtual void Dibujar(Ventana*)`
	`virtual void Limites(Rect&)`
detectar cortes	`virtual bool Interseca(const Punto&)`
estructura	`virtual void Insertar(Glifo*, int)`
	`virtual void Borrar(Glifo*)`
	`virtual Glifo* Hijo(int)`
	`virtual Glifo* Padre()`

Los glifos tienen tres responsabilidades básicas: (1) saber cómo dibujarse a sí mismos, (2) qué espacio ocupan y (3) cuáles son sus hijos y su padre.

Las subclases de Glifo redefinen la operación Dibujar para visualizarse en una ventana, pasándoles en la llamada a Dibujar una referencia al objeto Ventana. La clase **Ventana** define operaciones gráficas para visualizar texto y figuras básicas en una ventana de la pantalla. Una subclase **Rectángulo** puede redefinir Dibujar como sigue:

```
void Rectangulo::Dibujar (Ventana* v) {
  v->DibujarRect(_x0, _y0, _x1, _y1);
}
```

donde _x0, _y0, _x1, e _y1 son datos miembro de Rectángulo que definen dos esquinas opuestas del rectángulo. DibujarRectangulo es la operación de Ventana que hace que el rectángulo aparezca en la pantalla

Un glifo padre muchas veces necesitar saber cuánto espacio ocupa uno de sus hijos, por ejemplo para situarlo junto con otros glifos en una línea de manera que no se solapen (como se muestra en la Figura 2.2). La operación Limites devuelve el área rectangular que ocupa el glifo, es decir, las esquinas opuestas del rectángulo más pequeño que lo contiene. Las subclases de Glifo redefinen esta operación para devolver el área rectangular en que se inscriben.

La operación Interseca comprueba si un punto concreto corta al glifo. Cada vez que el usuario hace clic en algún punto del documento, Lexi llama a esta operación para determinar qué figura (o estructura de figuras) está situada bajo el cursor del ratón. La clase Rectángulo redefine esta operación para calcular la intersección del rectángulo y el punto especificado.

Puesto que los glifos pueden tener hijos, necesitamos una interfaz común para añadir, borrar y acceder a dichos hijos. Por ejemplo, los hijos de una Fila son los glifos que ésta coloca en una fila. La operación **Insertar** inserta un glifo en la posición especificada por un índice entero [15]. La operación **Borrar** borra la figura especificada en caso de que realmente se trate de un hijo.

La operación Hijo devuelve el hijo de la posición indicada, si es que éste existe. Los glifos que, como Fila, pueden tener hijos, deberían usar la operación Hijo internamente, en vez de acceder directamente a la estructura de datos donde se encuentran éstos. De esa manera no habrá que modificar operaciones como Dibujar que iteran a través de los hijos cuando se cambie la estructura de datos de, por ejemplo, un array a una lista enlazada. De manera similar. Padre proporciona una interfaz estándar al padre de glifo, si es que existe. En Lexi los glifos tienen una referencia a su padre, y su operación Padre simplemente devuelve dicha referencia.

PATRÓN COMPOSITE

La composición recursiva no sólo es útil para los documentos. También la podemos usar para representar cualquier estructura jerárquica potencialmente compleja. El patrón Composite (151)

representa la esencia de la composición recursiva en términos de orientación a objetos. Éste podría ser un buen momento para acudir a ese patrón y estudiarlo, volviendo a este escenario cuando sea necesario.

2.3. FORMATEADO

Ya nos hemos puesto de acuerdo sobre el modo de *representar* la estructura física de un documento. Ahora necesitamos saber cómo construir una estructura física *en particular*, es decir, una que corresponda a un documento concreto, correctamente formateado. La representación y el formateado son diferentes: la capacidad de representar la estructura física del documento no nos dice cómo llegar a una estructura concreta. Esta responsabilidad recae principalmente en Lexi. Es él quien debe separar el texto en líneas, las líneas en columnas y así sucesivamente, teniendo en cuenta los deseos del usuario. Por ejemplo, éste puede querer variar la anchura de los márgenes, el sangrado y las tabulaciones, interlineado sencillo o doble, y probablemente muchas otras restricciones de formateado[16]. El algoritmo de formateado de Lexi debe tener en cuenta todas estas cosas.

Tabla 2.2 Interfaz de un componedor básico

Responsabilidades	Operaciones
qué formatear	`void EstablecerComposicion(Composicion*)`
cuándo formatear	`virtual void Componer()`

Por cierto, restringiremos el término “formateado” para referirnos a la separación de los glifos en líneas. De hecho, usaremos los términos “formateado” y “separación de líneas” como sinónimos. Las técnicas que veremos se aplican por igual a la separación de las líneas en columnas y a la separación de columnas en páginas.

ENCAPSULACIÓN DEL ALGORITMO DE FORMATEADO

El proceso de formateado, con todos sus detalles y restricciones, no

es fácil de automatizar. Hay muchas aproximaciones al problema, y la gente ha ideado una variedad de algoritmos de formateado con diferentes puntos fuertes y débiles. Dado que Lexi es un editor WYSIWYG, una cuestión importante a tener en cuenta es el equilibrio entre calidad y velocidad de formateado. Generalmente queremos una buena respuesta del editor sin sacrificar la apariencia del documento. Este equilibrio depende de muchos factores, no todos los cuales pueden determinarse en tiempo de compilación. Por ejemplo, el usuario puede tolerar una respuesta ligeramente más lenta si a cambio obtiene un formateado mejor. Esto puede hacer que sea más apropiado un algoritmo de formateado distinto del original. Otro equilibrio, más orientado a la implementación, es el que se puede establecer entre la velocidad de formateado y los requisitos de almacenamiento: tal vez sea posible reducir el tiempo de formateado guardando más información.

Como los algoritmos de formateado tienden a ser complicados, también es deseable mantenerlos contenidos o —mejor aún— que sean completamente independientes de la estructura del documento. Idealmente, podríamos añadir una nueva subclase de Glifo sin depender del algoritmo de formateado. A la inversa, añadir un nuevo algoritmo de formateado no debería requerir modificar los glifos existentes.

Estas características sugieren que deberíamos diseñar Lexi para que sea fácil cambiar el algoritmo de formateado, al menos en tiempo de compilación, si no en tiempo de ejecución. Podemos aislar el algoritmo y a la vez hacer que sea fácilmente reemplazable encapsulándolo en un objeto. Más concretamente, definiremos una jerarquía de clases separada para los objetos que encapsulan algoritmos de formateado. La raíz de la jerarquía definirá una interfaz que soporta una gran variedad de algoritmos de formateado, y cada subclase implementará la interfaz para llevar a cabo un algoritmo concreto. Luego podemos introducir una subclase de Glifo que estructurará sus hijos automáticamente usando un objeto algoritmo dado.

COMPONEDOR Y COMPOSICIÓN

Definiremos una clase Componedor para los objetos que pueden encapsular un algoritmo de formateado. La interfaz (Tabla 2.2)

permite que el componedor sepa *qué* glifos formatear y *cuándo* hacer el formateado. Los glifos que formatea son los hijos de una subclase especial de Glifo llamada Composición, que obtiene una instancia de una subclase Componedor (especializada para un algoritmo concreto de separación de líneas) cuando se crea, y le ordena al componedor Componer sus glifos cuando sea necesario (por ejemplo, cuando el usuario cambia un documento). La Figura 2.5 muestra las relaciones entre las clases Composición y Componedor.

Un objeto Composición sin formatear sólo contiene los glifos visibles que constituyen el contenido básico del documento. No contiene glifos que determinan la estructura física del documento, tales como Fila y Columna. La composición se encuentra en este estado justo después de ser creada e inicializada con los glifos que debería formatear. Cuando necesita ser formateada, llama a su operación Componer. A su vez, el componedor itera a través de los hijos de la composición e inserta nuevos glifos Fila y Columna en función de su algoritmo de separación de líneas[17]. La Figura 2.6 muestra la estructura de objetos resultante. Los glifos que han sido creados e insertados en la estructura de objetos por el componedor aparecen con fondo gris en la figura.

Figura 2.5: Relaciones entre las clases Composición y Componedor

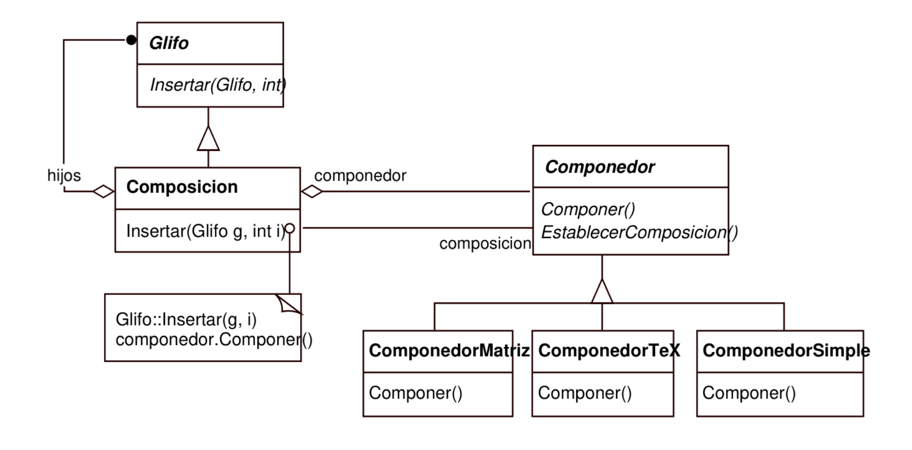
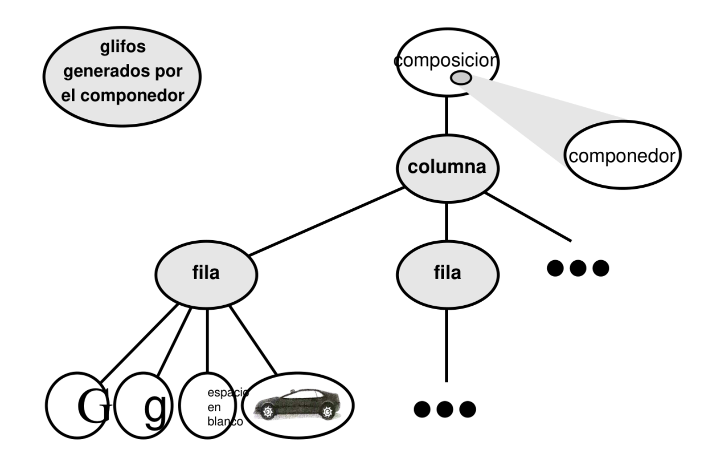


Figura 2.6: Estructura de objetos que muestra el interlineado

dirigido por un componedor



Cada subclase de Componedor puede implementar un algoritmo diferente de separación de líneas. Por ejemplo, un ComponedorSimple puede hacer una pasada rápida sin preocuparse de cuestiones como el “color” del documento. Un buen color significa que el documento tiene una distribución uniforme de texto y espacios en blanco. Un ComponedorTeX implementaría el algoritmo TeX [Knu84] completo, que tiene en cuenta cosas como el color a cambio de mayores tiempos de formateado.

Tener dos clases distintas Componedor y Composición garantiza una gran separación entre el código correspondiente a la estructura física del documento y el que lleva a cabo diferentes algoritmos de formateado. Podemos añadir nuevas subclases de Componedor sin tocar las clases de los glifos, y viceversa. De hecho, podemos cambiar el algoritmo de separación de líneas en tiempo de ejecución añadiendo una única operación EstablecerComponedor a la interfaz básica de un glifo Composición.

PATRÓN STRATEGY (ESTRATEGIA)

El propósito del patrón Strategy (289) es encapsular un algoritmo en un objeto. Los participantes principales en este patrón son los objetos Estrategia (que encapsulan diferentes algoritmos) y el contexto en el que éstos operan. Los componedores son estrategias que encapsulan diferentes algoritmos de formateado. Una

composición representa el contexto de estas estrategias.

La clave para aplicar el patrón Strategy es diseñar interfaces para la estrategia y su contexto que sean lo suficientemente generales como para permitir diversos algoritmos. No deberíamos tener que cambiar la interfaz de la estrategia o de su contexto para dar cabida a un nuevo algoritmo. En nuestro ejemplo, la capacidad de la interfaz básica de un Glifo para acceder a sus hijos, insertarlos y borrarlos es lo bastante general como para que las subclases de Componedor cambien la estructura física del documento, independientemente del algoritmo que usen para hacerlo. Así mismo, la interfaz de Componedor ofrece a las composiciones todo lo que necesitan para empezar el formateado.

2.4. ADORNAR LA INTERFAZ DE USUARIO

Hemos considerado dos adornos para la interfaz de usuario de Lexi. El primero añade un borde alrededor del área de edición de texto para delimitar la página. El segundo añade barras de desplazamiento que permiten al usuario ver diferentes partes de la página. Para que sea fácil añadir y borrar estos adornos (especialmente en tiempo de ejecución) no deberíamos usar la herencia para añadirlos a la interfaz de usuario. Conseguiremos la máxima flexibilidad si otros objetos de la interfaz de usuario ni siquiera son conscientes de la presencia de esos adornos, lo que nos permitirá añadirlos y borrarlos sin cambiar otras clases.

RECINTO TRANSPARENTE [18]

Desde un punto de vista programático, mejorar el aspecto de la interfaz de usuario implica ampliar el código existente. Usar la herencia para realizar dicha extensión impide cambiar los adornos en tiempo de ejecución, pero este enfoque presenta además otro problema igual de importante, como es la explosión de clases a la que puede dar lugar.

Podríamos añadir un borde a Composición heredando de ella para obtener clase ComposicionConBorde, o añadir una interfaz de

desplazamiento de la misma manera para dar lugar a una clase ComposicionDesplazable. Si quisiéremos tanto las barras de desplazamiento como un borde, podríamos tener una ComposicionDesplazableConBorde, y así sucesivamente. Llevándolo al extremo, acabaríamos por tener una clase para cada posible combinación, una solución que rápidamente se convierte en impracticable en cuanto aumenta el número de adornos.

La composición de objetos ofrece un mecanismo de ampliación más viable y extensible. Pero ¿qué objetos componemos? Puesto que sabemos que estamos adornando un glifo existente, podríamos hacer que el propio adorno fuese un objeto (por ejemplo, una instancia de la clase **Borde**). Eso nos da dos candidatos para la composición: el glifo y el borde. El paso siguiente es decidir quién contiene a quién. Podríamos hacer que el borde contenga al glifo, lo que tiene sentido dado que el borde rodeará al glifo en la pantalla. O podríamos hacer lo contrario —poner el borde dentro del glifo—, pero entonces tenemos que hacer modificaciones a las correspondientes subclases de Glifo para que sean conscientes del borde. Nuestra primera elección, introducir el glifo en el borde, mantiene el código que se encarga de dibujar el borde enteramente en la clase Borde, dejando las otras clases intactas.

¿A qué se parece la clase Borde? El hecho de que los bordes tengan representación visual sugiere que deberían ser glifos: es decir, que el borde debería ser una subclase de Glifo. Pero hay una razón más convincente para hacer esto: los clientes deberían tratar a todos los glifos de manera uniforme, independientemente de que estos tengan bordes o no. Cuando los clientes le dicen a un glifo normal, sin borde, que se dibuje, éste debería hacerlo sin ningún tipo de adorno. Si ese glifo está dentro de un borde, los clientes no deberían tener que tratar de modo diferente al borde que lo contiene, sino que simplemente le dirán que se dibuje, igual que se lo dijeron antes al glifo normal. Esto implica que la interfaz de Borde debe coincidir con la de Glifo, por lo que heredamos Borde de Glifo para garantizar dicha relación.

Todo esto nos lleva al concepto de **recinto transparente**, que aúna las nociones de (1) composición de un único hijo (o **componente** individual) y (2) interfaces compatibles. Generalmente los clientes no saben si están tratando con un

componente o con su **recinto** (es decir, su padre), sobre todo si éste no hace más que delegar en su componente todas sus operaciones. Pero el recinto también puede *aumentar* el comportamiento del componente haciendo tareas propias antes y después de delegar una operación. Así mismo, puede también añadir estado al componente. A continuación veremos cómo.

MONOGLIFO

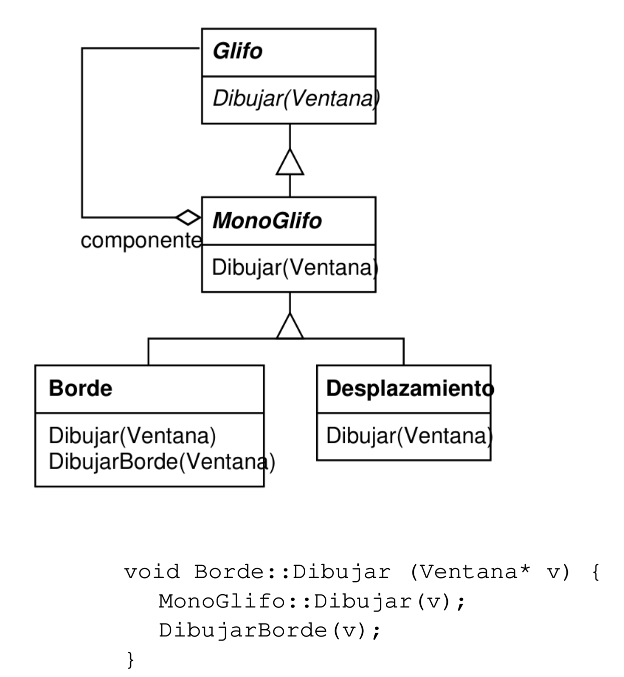
Podemos aplicar el concepto de recinto transparente a todos los glifos que pueden adornar a otros. Para concretar este concepto, definiremos una subclase de Glifo llamada **MonoGlifo**, que servirá de clase abstracta para los "glifos de adorno", como Borde (*véase* la Figura 2.7). MonoGlifo guarda una referencia a un componente y reenvía todas las peticiones a él.

Eso hace que, por omisión, MonoGlifo sea totalmente transparente a los clientes. Por ejemplo, MonoGlifo implementa la operación Dibujar como sigue:

```
void MonoGlifo::Dibujar (Ventana* v) {
  _componente->Dibujar(v);
}
```

Las subclases de MonoGlifo reimplementan al menos una de estas operaciones de reenvío. Borde::Dibujar, por ejemplo, primero invoca a la operación de la clase padre del componente, MonoGlifo::Dibujar, para que el componente haga su parte, es decir, dibujar todo menos el borde. Después, Borde::Dibujar dibuja el borde llamando a una operación privada DibujarBorde, cuyos detalles omitiremos:

Figura 2.7: Relaciones de la clase MonoGlifo




```
void Borde::Dibujar (Ventana* v) {
   MonoGlifo::Dibujar(v);
   DibujarBorde(v);
}
```

Nótese cómo Borde::Dibujar efectivamente *extiende* la operación de la clase padre para dibujar el borde, en vez de simplemente *reemplazar* la operación de la clase padre, lo que omitiría la llamada a MonoGlifo::Dibujar.

Otra subclase de MonoGlifo aparece en la Figura 2.7. **Desplazamiento** [19] es un MonoGlifo que dibuja a su componente en diferentes ubicaciones basándose en la posición de las dos barras de desplazamiento que le añade como adornos. El Desplazamiento, al dibujar su componente, le dice al sistema de gráficos que lo ajuste a sus límites. Al recortar las partes del componente que están desplazadas fuera de la vista se evita que aparezcan en la pantalla.

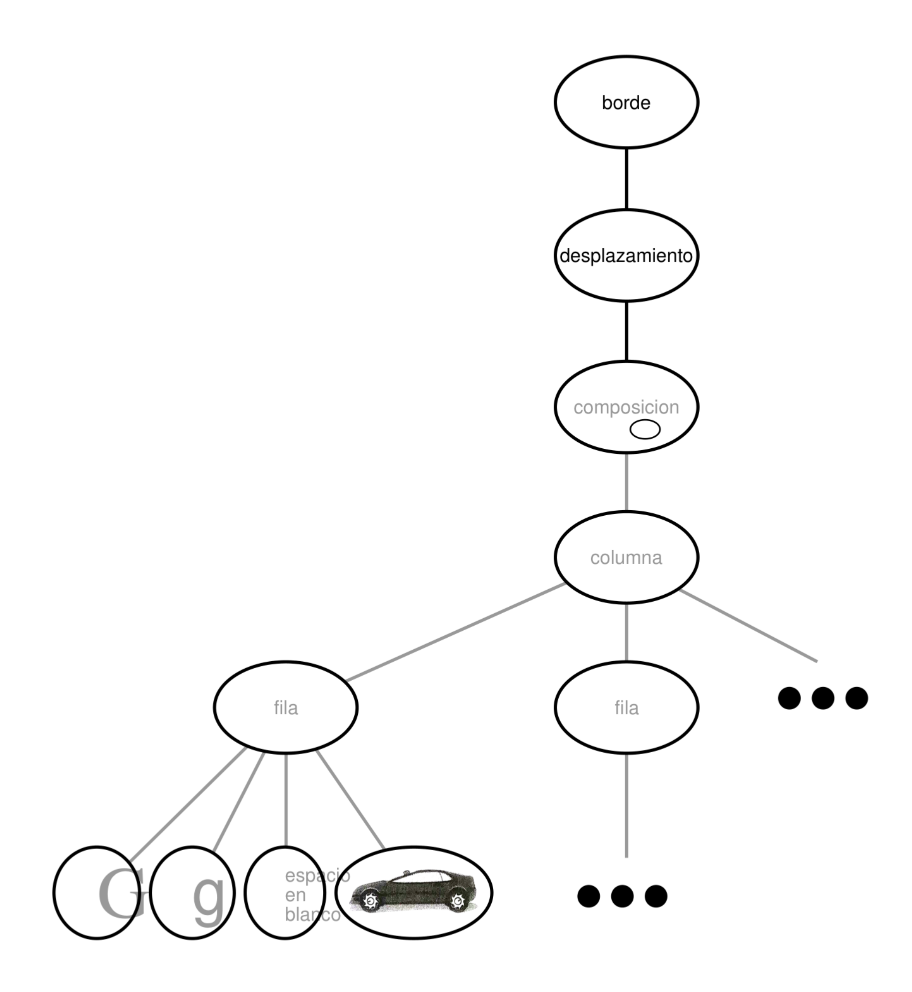
Ahora ya contamos con todas las piezas necesarias para añadir un borde y una interfaz de desplazamiento al área de edición de texto de Lexi. Para ello, compondremos la instancia existente de Composición en una instancia de Desplazamiento, para añadir la capacidad de desplazamiento, y el resultado lo compondremos en una instancia Borde. La estructura de objetos resultante es la que se

muestra en la Figura 2.8.

Nótese que podemos cambiar el orden de la composición, poniendo la composición con el borde en la instancia de Desplazamiento. En ese caso el borde se desplazaría con el texto, lo cual puede ser o no deseable. La cuestión es que el recinto transparente hace que sea fácil experimentar con diferentes alternativas, liberando a los clientes del código de adorno.

Es también importante notar cómo el borde compone un solo glifo, y no dos o más. Esto es diferente de las composiciones que definimos antes, en las que los objetos padre podían tener un número arbitrario de hijos. Aquí, poner un borde alrededor de algo implica que ese “algo” es singular. Podríamos dar un significado a adornar más de un objeto a la vez, pero entonces tendríamos que mezclar muchos tipos de composición con la noción de adorno: adorno de fila, de columna, etcétera. Eso no nos sería de ayuda, puesto que ya tenemos clases que hacen ese tipo de composiciones. Así que es mejor usar las clases existentes para la composición y añadir nuevas clases para adornar el resultado. Mantener los adornos independientes de otros tipos de composición simplifica las clases que se encargan de los adornos y reduce su número. También nos evita tener que replicar la funcionalidad de composición existente.

Figura 2.8: Estructura de objetos con los adornos



PATRÓN DECORATOR (DECORADOR)

El patrón Decorator (161) expresa las relaciones de clases y objetos que permiten decorar la interfaz mediante los recintos transparentes. El término “decoración” en realidad tiene un significado más amplio que el que hemos considerado aquí. En el patrón Decorator, la decoración se refiere a cualquier cosa que añada responsabilidades a un objeto. Podemos pensar, por ejemplo, en adornar un árbol de sintáctico abstracto con acciones semánticas, un autómata de estados finitos con nuevas transiciones, o una red de objetos persistentes con nuevas etiquetas de atributos. El patrón Decorator generaliza el enfoque que hemos seguido en Lexi para hacerlo aplicable en otros ámbitos.

2.5. PERMITIR MÚLTIPLES ESTÁNDARES DE INTERFAZ DE USUARIO[20]

Conseguir portabilidad entre plataformas hardware y software es uno de los principales problemas del diseño de sistemas. Adaptar Lexi a una nueva plataforma no debería requerir una revisión a fondo, o de lo contrario no merecería la pena la adaptación. Debemos hacer que portar el sistema sea lo más fácil posible.

Un obstáculo para conseguir la portabilidad es la variedad de estándares de interfaz de usuario existentes, pensados para imponer una uniformidad entre las distintas aplicaciones. Estos estándares definen directrices sobre la apariencia de las aplicaciones y cómo éstas reaccionan ante el usuario. Aunque los estándares existentes no son tan diferentes entre sí, lo cierto es que nadie los confundirá —las aplicaciones para Motif no tienen exactamente el mismo aspecto que sus homologas en otras plataformas, y viceversa—. Una aplicación que se ejecuta en más de una plataforma debe ajustarse a la guía de estilo de interfaces de usuario de cada una de ellas.

Nuestros objetivos de diseño son hacer que Lexi se ajuste a múltiples estándares de interfaz de usuario y que sea fácil admitir otros nuevos a medida que vayan surgiendo (como invariablemente sucederá). También queremos que nuestro diseño tenga la máxima flexibilidad: que se pueda cambiar la interfaz de usuario de Lexi en tiempo de ejecución.

ABSTRAYENDO LA CREACIÓN DE OBJETOS

Cualquier cosa que veamos y con la que interactuemos en la interfaz de Lexi es un glifo combinado con otros glifos invisibles, como Fila y Columna. Los glifos invisibles componen a otros visibles, como Boton y Carácter, y los ubican correctamente. Las guías de estilo tienen mucho que decir acerca de la apariencia y el comportamiento de los llamados “útiles”[21], otro término con el que se designa a los glifos visibles, como botones, barras de desplazamiento y menús, que controlan los elementos de una interfaz de usuario. Los útiles podrían usar glifos más sencillos como caracteres, círculos, rectángulos y polígonos para presentar

los datos.

Supondremos que tenemos dos conjuntos de clases de glifos de útiles con los que implementar múltiples estándares de interfaz de usuario:

1. Un conjunto de subclases abstractas de Glifo para cada categoría de glifo de útil. Por ejemplo, una clase abstracta BarraDeDesplazamiento aumentará la interfaz de un glifo básico para añadir operaciones generales de desplazamiento; Boton es una clase abstracta que añade operaciones relacionadas con los botones; etcétera.
2. Un conjunto de subclases concretas para cada subclase abstracta, las cuales implementan diferentes estándares de interfaz de usuario. Por ejemplo, BarraDeDesplazamiento podría tener las subclases BarraDeDesplazamientoMotif y BarraDeDesplazamientoPM, que implementan barras de desplazamiento para Motif y Presentation Manager, respectivamente.

Lexi debe distinguir entre los glifos de útiles para diferentes estilos de interfaces de usuario. Por ejemplo, cuando necesita colocar un botón en su interfaz, debe crear una instancia de una subclase de Glifo con el estilo adecuado de botón (BotonBoton, BotonPM, BotonMac, etc.).

Está claro que la implementación de Lexi no puede hacer esto directamente (por ejemplo, mediante una llamada a un constructor de C++). Eso fijaría en el código el botón de un estilo concreto, haciendo imposible seleccionar el estilo en tiempo de ejecución. También tendríamos que localizar cada llamada a dicho constructor y cambiarlas para portar Lexi a otra plataforma. Y los botones son sólo uno de los muchos útiles de los que consta la interfaz de usuario de Lexi. Plagar nuestro código con llamadas a los constructores de las clases de una interfaz de usuario concreta da como resultado una pesadilla de mantenimiento, donde pasar por alto uno solo de estos constructores hará que nos encontremos un menú de Motif en medio de nuestra aplicación para Mac.

Lexi necesita un modo de determinar los estándares de interfaz de usuario a los que está dirigido, para así crear los útiles apropiados. No sólo debemos evitar hacer llamadas explícitas a

constructores: también debemos ser capaces de reemplazar un conjunto completo de útiles fácilmente. Podemos lograr ambas cosas *abstrayendo el proceso de creación de objetos*. Un ejemplo ilustrará qué queremos decir con esto.

CLASES FÁBRICA Y PRODUCTO

Normalmente podemos crear una instancia de una barra de desplazamiento de Motif con el siguiente código C++:

```
BarraDeDesplazamiento* barra = new
BarraDeDesplazamientoMotif;
```

Ésta es la clase de código que debemos evitar si queremos minimizar las dependencias de la interfaz de usuario en Lexi. Supongamos que inicializamos barra como sigue:

```
BarraDeDesplazamiento* barra =
               fabricaIGU-
>CrearBarraDeDesplazamiento();
```

donde fabricaIGU es una instancia de la clase **FabricaMotif**. CrearBarraDeDesplazamiento devuelve una instancia de la subclase correcta de BarraDeDesplazamiento para la interfaz de usuario deseada, en este caso Motif. En cuanto a los clientes, el efecto es el mismo que llamar directamente al constructor de BarraDeDesplazamientoMotif, pero con una diferencia crucial: ya no hay nada en el código que haga referencia a Motif por su nombre. El objeto fabricaIGU abstrae el proceso de creación no sólo de barras de desplazamiento para Motif, sino de barras de desplazamiento para *cualquier* estándar de interfaz de usuario. Y tampoco se limita a producir barras de desplazamiento, sino que puede fabricar una amplia variedad de útiles, incluyendo barras de desplazamiento, botones, campos de entrada, menús, etcétera.

Todo esto es posible porque FabricaMotif es una subclase de **FabricaIGU**, una clase abstracta que define una interfaz general

para crear glifos de útiles. Incluye operaciones como CrearBarraDeDesplazamiento y CrearBoton para diferentes tipos de glifos de útiles (*widgets*). Las subclases de FabricaIGU implementan estas operaciones para devolver glifos tales como BarraDeDesplazamientoMotif y BotonPM que implementan una interfaz de usuario en concreto. La Figura 2.9 muestra la jerarquía de clases resultante para los objetos fabricaIGU.

Decimos que las fábricas crean objetos **producto**. Además, los productos creados por una fábrica se relacionan entre sí; en este caso, los productos son todos útiles (*widgets*) con el mismo estilo de interfaz de usuario. La Figura 2.10 muestra algunas de las clases necesarias para que las fábricas trabajen con glifos de útiles.

La última cuestión a la que tenemos que responder es de dónde viene la instancia de FabricaIGU. La respuesta es que de cualquier sitio. La variable fabricaIGU podría ser un miembro estático de una clase conocida, o incluso una variable local si toda la interfaz de usuario se creara dentro de la misma clase o función. Hay un patrón de diseño, el Singleton (119), que permite tratar con objetos conocidos de los que sólo hay una instancia, como es este caso. Lo importante es inicializar fabricaIGU en un punto del programa que esté *antes* de que se use por primera vez para crear útiles, pero *después* de que esté claro qué interfaz de usuario se desea.

Figura 2.9: Jerarquía de clases de FabricaIGU

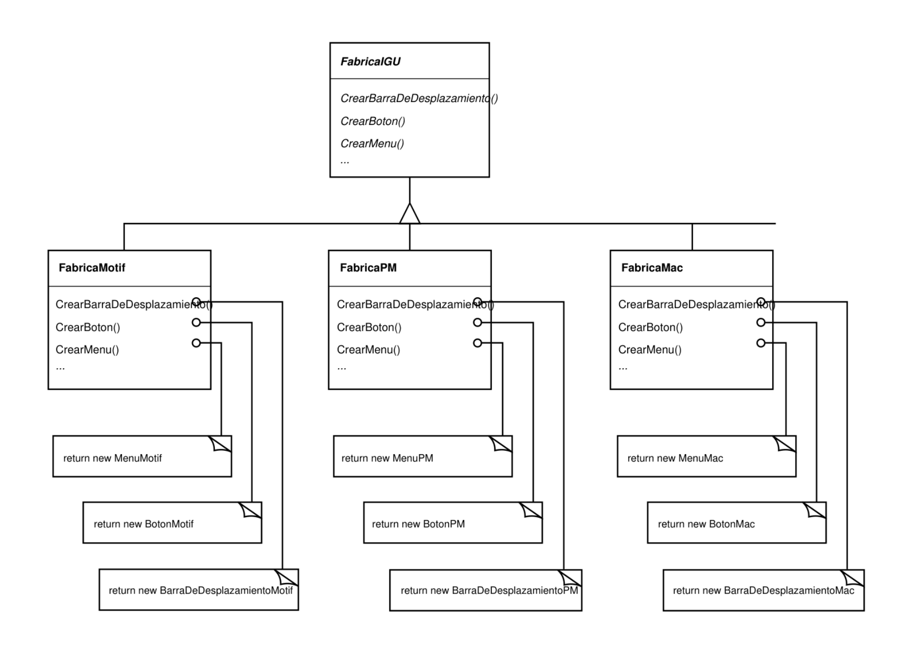
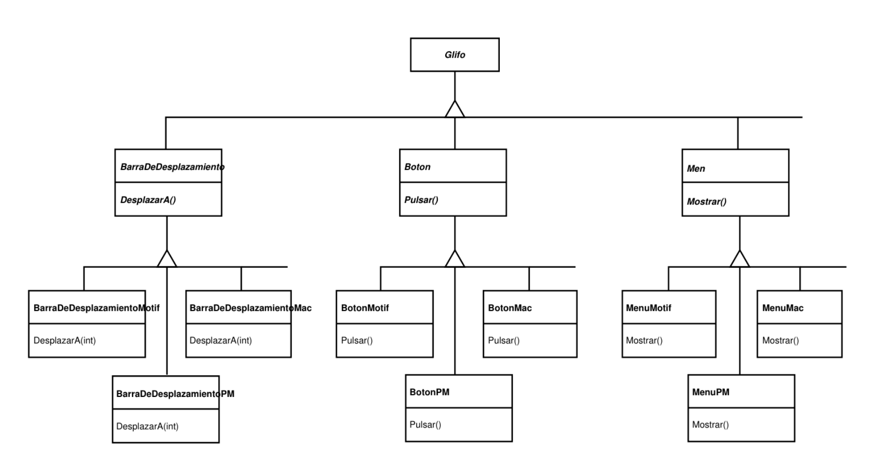


Figura 2.10: Clases abstractas de producto y subclases concretas



Si la interfaz de usuario se conoce en tiempo de compilación, fabricaIGU puede ser inicializada con una simple asignación de una nueva instancia de una fábrica al comienzo del programa:

```
FabricaIGU* fabricaIGU = new FabricaMotif;
```

Si el usuario puede especificar la interfaz de usuario con un nombre al inicializar el programa, entonces el código para crear la fábrica podría ser el siguiente:

```
FabricaIGU* fabricaIGU;
const        char*        nombreEstilo        =
getenv("INTERFAZ_DE_USUARIO");
   // el usuario o el entorno lo proporcionan
al principio

if (strcmp(nombreEstilo, "Motif") == 0) {
   fabricaIGU = new FabricaMotif;

}      else      if      (strcmp(nombreEstilo,
"Presentation_Manager") == 0) {
   fabricaIGU = new FabricaPM;

} else {
   fabricaIGU = new FabricaIGUPredeterminada;
}
```

Hay formas más sofisticadas de seleccionar la fábrica en tiempo de ejecución. Por ejemplo, podríamos tener un registro que hiciera corresponder cadenas de texto con objetos fábrica. Eso nos permitiría registrar instancias de nuevas subclases de fábricas sin modificar el código existente, como requiere el enfoque anterior. Y no tenemos que enlazar todas las fábricas dependientes de la plataforma en la aplicación. Esto es importante, porque podría no ser posible enlazar una FabricaMotif en una plataforma que no admita Motif.

Pero la clave es que una vez que hemos configurado la aplicación con el objeto fábrica adecuado, a partir de entonces se activa su estilo de interfaz de usuario. Si cambiamos de idea, podemos reinicializar fabricaIGU con una fábrica para otro estilo de interfaz de usuario, y reconstruir entonces la interfaz. Independientemente de cómo y cuándo decidamos inicializar fabricaIGU, sabemos que una vez que lo hagamos la aplicación puede crear la interfaz de usuario apropiada sin ninguna modificación.

PATRÓN ABSTRACT FACTORY (FÁBRICA ABSTRACTA)

Las fábricas y los productos son los principales participantes del patrón Abstract Factory (79). Este patrón muestra cómo crear familias de objetos producto relacionados sin instanciar las clases directamente. Es apropiado sobre todo cuando el número y tipo de los objetos producto permanece constante y hay diferencias entre las distintas familias de productos. Podemos elegir entre las distintas familias creando una instancia de una fábrica concreta y usándola de forma consistente para crear productos a partir de entonces. También podemos cambiar familias completas de productos reemplazando la fábrica concreta con una instancia de otra diferente. La importancia que el patrón Abstract Factory da a las *familias* de productos diferencia a éste de otros patrones de creación, que comprenden un solo tipo de objetos producto.

2.6. PERMITIR MÚLTIPLES SISTEMAS DE VENTANAS

El estilo de la interfaz de usuario es sólo una de las muchas cuestiones que tienen que ver con la portabilidad. Otra es el entorno de ventanas en el que se ejecuta Lexi. Un sistema de ventanas crea la ilusión de ventanas solapadas en una pantalla de mapa de bits. También controla el espacio en pantalla para las ventanas y dirige hacia ellas la entrada del teclado y del ratón. Actualmente existen varios sistemas de ventanas importantes y, en gran medida, incompatibles (por ejemplo, Macintosh, Presentation Manager, Windows, X). Nos gustaría que Lexi se ejecutase en tantos de ellos como fuese posible, por la misma razón por la que permitimos múltiples estándares de interfaz de usuario.

¿PODEMOS USAR UNA FABRICA ABSTRACTA?

A primera vista esto puede parecer otra ocasión para aplicar el patrón Abstract Factory. Pero las restricciones en cuanto a la portabilidad del sistema de ventanas difieren significativamente de las de la independencia de la interfaz de usuario.

En aquel caso, cuando aplicábamos el patrón Abstract Factory estábamos asumiendo que definiríamos clases concretas de útiles para cada estándar de interfaz de usuario. Eso significaba que podríamos derivar cada producto concreto para un estándar en particular (por ejemplo, BarraDeDesplazamientoMotif y BarraDeDesplazamientoMac) de una clase de producto abstracta (por ejemplo, BarraDeDesplazamiento). Pero supongamos que ya tenemos varias jerarquías de clases de diferentes vendedores, una por cada estándar de interfaz de usuario. Por supuesto, es altamente improbable que estas jerarquías sean en modo alguno compatibles. Así que no tendremos una clase de producto abstracta por cada tipo de útil (BarraDeDesplazamiento, Boton, Menu, etc.) y el patrón Abstract Factory no funcionará sin esas clases cruciales. Tenemos que hacer que las diferentes jerarquías de útiles se adhieran a un conjunto común de interfaces de productos abstractas. Sólo entonces podríamos declarar las operaciones Crear… en nuestra interfaz de fábrica abstracta.

En el caso de los útiles resolvimos este problema desarrollando nuestras propias clases abstractas y concretas de productos. Ahora nos enfrentamos a un problema similar cuando intentamos que Lexi funcione en diferentes sistemas de ventanas; a saber, que los diferentes sistemas de ventanas tienen interfaces de programación incompatibles. Esta vez, no obstante, las cosas son un poco más difíciles, porque no podemos permitimos implementar nuestro propio sistema de ventanas no estándar.

Pero nos salva que las interfaces de los sistemas de ventanas, al igual que las de los diferentes estándares de interfaces de usuario, no son radicalmente diferentes unas de otras, porque por lo general todos los sistemas de ventanas hacen las mismas cosas. Necesitamos un conjunto uniforme de abstracciones de ventanas que nos permitan tomar diferentes implementaciones de sistemas de ventanas y poner cualquiera de ellas bajo una interfaz común.

ENCAPSULAR DEPENDENCIAS DE IMPLEMENTACIÓN

En la Sección 2.2 introducimos una clase Ventana para mostrar un glifo o una estructura de glifos en pantalla. No especificamos el sistema de ventanas con el que trabajaba este objeto, porque la verdad es que no proviene de ningún sistema de ventanas. La clase

Ventana encapsula las operaciones que suelen hacer las ventanas en los diferentes sistemas de ventanas:

- Proporcionan operaciones para dibujar formas geométricas básicas.
- Pueden minimizarse y maximizarse a sí mismas.
- Pueden cambiar su tamaño.
- Pueden (re)dibujar sus contenidos a petición, por ejemplo cuando son maximizadas o cuando se muestra una parte que permanecía oculta de su espacio en pantalla.

La clase Ventana debe abarcar la funcionalidad de las ventanas de diferentes sistemas de ventanas. Consideremos dos filosofías extremas:

1. *Intersección de funcionalidad.* La clase Ventana proporciona sólo la funcionalidad que es común a *todos* los sistemas de ventanas. El problema de este enfoque es que nuestra interfaz de Ventana acaba siendo sólo tan potente como el sistema de ventanas con menos capacidades. No podemos aprovechamos de características más avanzadas incluso aunque la mayoría (pero no todos) de los sistemas de ventanas las admitan.
2. *Unión de funcionalidad.* Crear una interfaz que incorpora las capacidades de *todos* los sistemas existentes. El problema es que la interfaz resultante puede ser gigantesca e incoherente. Además, tendremos que cambiarla (y a Lexi, que depende de ella) cada vez que un vendedor revise la interfaz de su sistema de ventanas.

Ninguno de los dos extremos es una solución viable, de manera que nuestro diseño se situará en algún punto intermedio entre ambos. La clase Ventana proveerá una interfaz adecuada que tenga las características más populares de los sistemas de ventanas. Como Lexi tratará directamente con ella, la clase Ventana debe incluir también un conjunto básico de operaciones gráficas que permita a los glifos dibujarse a sí mismos en una ventana. La Tabla 2.3 muestra un ejemplo de las operaciones de la interfaz de la clase Ventana.

Tabla 2.3: Interfaz de la clase Ventana

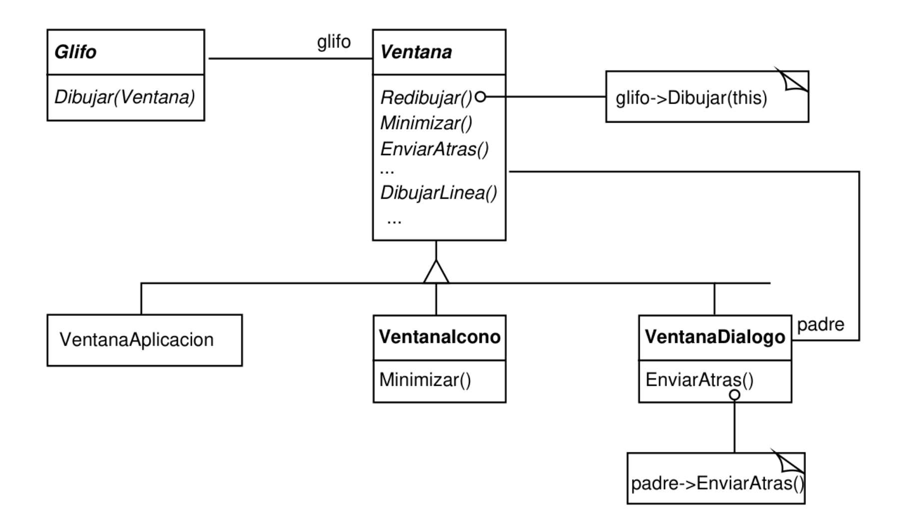
Responsabilidades	Operaciones

gestión de ventanas	`virtual void Redibujar()`
	`virtual void TraerAdelante()`
	`virtual void EnviarAtras()`
	`virtual void Minimizar()`
	`virtual void Maximizar()`
	…

gráficos	`virtual void DibujarLinea(…)`
	`virtual void DibujarRect(…)`
	`virtual void DibujarPoligono(…)`
	`virtual void DibujarTexto(…)`
	…

Ventana es una clase abstracta. Sus subclases concretas admiten los diferentes tipos de ventanas con las que tratarán los usuarios. Por ejemplo, las ventanas de aplicación, los iconos y los diálogos de avisos, son todos ellos ventanas, pero tienen comportamientos diferentes. Así que podemos definir subclases como VentanaDeAplicacion, VentanaIcono y VentanaDialogo para representar tales diferencias. La jerarquía de clases resultante le da a las aplicaciones como Lexi una abstracción de las ventanas uniforme e intuitiva, que no depende del sistema de ventanas de ningún vendedor en particular.

Ahora que hemos definido una interfaz de ventana para que Lexi trabaje con ella, ¿cuál es el papel del sistema de ventanas real de una plataforma concreta? Dado que no estamos implementando nuestro propio sistema de ventanas, nuestra abstracción de ventana deberá ser implementada en algún lugar en términos de lo que proporciona el sistema de ventanas nativo. ¿Dónde reside esa implementación?



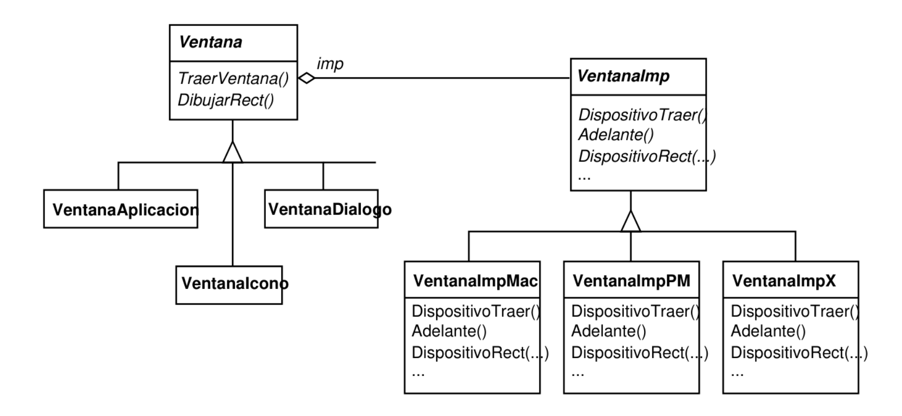
Un enfoque es implementar múltiples versiones de la clase Ventana y sus subclases, una para cada plataforma de ventanas. Tendríamos que elegir la versión a usar cuando construyéramos Lexi para una plataforma dada. Pero imagine los dolores de cabeza que daría el mantenimiento si tuviésemos que estar pendientes de múltiples clases, todas llamadas “Ventana”, pero cada una implementada en un sistema de ventanas diferente. Otra alternativa sería crear subclases dependientes de la implementación para cada clase de la jerarquía de Ventana —y acabar teniendo otro problema de explosión de subclases como el que tuvimos cuando tratábamos de añadir adornos—. Ambas alternativas tienen sus desventajas: ninguna nos da la flexibilidad de cambiar el sistema de ventanas después de haber compilado el programa. De manera que tendremos que tener diferentes ejecutables.

Ninguna de estas opciones es muy atractiva, pero ¿qué otra cosa podemos hacer? Lo mismo que hicimos para el formateado y la decoración, es decir, *encapsular el concepto que varía*. Lo que cambia en este caso es la implementación del sistema de ventanas. Si encapsulamos la funcionalidad de un sistema de ventanas en un objeto, podemos implementar nuestra clase Ventana y sus subclases en términos de la interfaz de ese objeto. Además, si esa interfaz sirve para todos los sistemas de ventanas en los que estamos interesados, no tendremos que cambiar Ventana ni ninguna de sus subclases para permitir diferentes sistemas de ventanas. Podemos

configurar los objetos ventana para el sistema de ventanas que queramos simplemente pasándoles el objeto apropiado que lo encapsula. Incluso podemos configurar la ventana en tiempo de ejecución.

VENTANA Y VENTANAIMP

Definiremos una jerarquía de clases **VentanaImp** aparte, para ocultar las implementaciones de los diferentes sistemas de ventanas. VentanaImp es una clase abstracta para los objetos que encapsulan código dependiente del sistema de ventanas. Para que Lexi funcione en un sistema de ventanas concreto, configuramos cada objeto ventana con una instancia de la subclase de VentanaImp para ese sistema. El diagrama de la página siguiente muestra la relación entre las jerarquías de Ventana y VentanaImp: Al ocultar las implementaciones en las clases VentanaImp, evitamos contaminar las clases Ventana con dependencias del sistema de ventanas, lo que mantiene a la jerarquía de clases de Ventana relativamente pequeña y estable. Además, podemos ampliar fácilmente la jerarquía de implementación para admitir nuevos sistemas de ventanas.



Subclases de VentanaImp

Las subclases de VentanaImp convierten peticiones en operaciones específicas del sistema de ventanas. Piense en el ejemplo que empleamos en la Sección 2.2. Definíamos Rectángulo:: Dibujar en términos de la operación DibujarRect de la instancia de Ventana:

```
void Rectangulo::Dibujar (Ventana* v) {
  v->DibujarRect(_x0, _y0, _x1, _y1);
}
```

La implementación predeterminada de DibujarRect usa la operación abstracta para dibujar rectángulos declarada por VentanaImp:

```
void Ventana::DibujarRect (
  Coord x0, Coord y0, Coord x1, Coord y1
) {
  _imp->DispositivoRect(x0, y0, x1, y1);
}
```

donde _imp es una variable miembro de Ventana que guarda el objeto VentanaImp con el que se configuró la ventana. La implementación de la ventana está definida por la instancia de la subclase VentanaImp a la que apunta _imp. Para una VentanaImpX (esto es, una subclase de VentanaImp para el sistema de ventanas X), la implementación de DispositivoRect puede parecerse a

```
void VentanaImpX::DispositivoRect (
  Coord x0, Coord y0, Coord x1, Coord y1
) {
  int x = round(min(x0, x1));
  int y = round(min(y0, y1));
  int ancho = round(abs(x0 - x1));
  int alto = round(abs(y0 - y1));
  XDrawRectangle(_dpy,  winid,  cg,  x,  y,
ancho, alto);
}
```

DispositivoRect está así definido porque XDrawRectangle (la interfaz de X para dibujar un rectángulo) define un rectángulo en términos de su esquina inferior izquierda, su ancho y su alto. DispositivoRect debe calcular estos valores a partir de los que recibe. En primer lugar determina la esquina inferior izquierda (ya que (x0, y0) puede ser cualquiera de las cuatro esquinas del

rectángulo) y luego calcula el ancho y el alto.

VentanaImpPM (una subclase de VentanaImp para Presentation Manager) definiría DispositivoRect de esta otra forma:

```
void VentanaImpPM::DispositivoRect (
   Coord x0, Coord y0, Coord x1, Coord y1
) {
   Coord izquierda = min(x0, x1);
   Coord derecha = max(x0, x1);
   Coord inferior = min(y0, y1);
   Coord superior = max(y0, y1);

   PPOINTL punto[4];

   punto[0].x   =   izquierda;   punto[0].y   =
superior;
   punto[1].x    =    derecha;    punto[1].y    =
superior;
   punto[2].x    =    derecha;    punto[2].y    =
inferior;
   punto[3].x   =   izquierda;   punto[3].y   =
inferior;

   if (
      (GpiBeginPath(_hps, 1L) == false) ¦¦
      (GpiSetCurrentPosition(_hps,  &punto[3])
== false) ¦¦
      (GpiPolyLine(_hps,    4L,    punto)    ==
GPI_ERR0R) ¦¦
      (GpiEndPath(_hps) == false)
   ) {
      // informar del error

   } else {
      GpiStrokePath(_hps, 1L, 0L);
   }
}
```

¿Por qué es tan diferente de la versión para X? Bueno, PM no tiene una operación para dibujar rectángulos explícitamente, como sí tenía X. En lugar de eso, PM tiene una interfaz más general para especificar vértices de figuras formadas por varios segmentos (lo que se conoce como un ***path***) y para trazar o rellenar el área que encierran.

La implementación para PM de DispositivoRect difiere bastante de la de X, pero eso no importa. VentanaImp oculta las diferencias de las interfaces de los sistemas de ventanas tras una interfaz potencialmente grande pero estable. Eso permite que los creadores de las subclases de Ventana se centren en la abstracción de ventana y no en los detalles del sistema de ventanas. También nos permite añadir nuevos sistemas de ventanas sin afectar a las clases Ventana.

Configurar Ventanas con VentanaImps

Una cuestión clave que aún no hemos considerado es cómo se configura una ventana con la subclase correcta de VentanaImp. Dicho de otra forma, ¿cuándo se inicializa _imp, y quién sabe qué sistema de ventanas (y, consecuentemente, qué subclase de VentanaImp) se está usando? La ventana necesitará algún tipo de VentanaImp antes de que pueda hacer nada interesante.

Hay varias posibilidades, pero nos centraremos en una que hace uso del patrón Abstract Factory (79). Podemos definir una clase fábrica abstracta FabricaSistemaDeVentanas que proporciona una interfaz para crear diferentes tipos de implementaciones de objetos dependientes del sistema de ventanas:

```
class FabricaSistemaDeVentanas {
public:
   virtual VentanaImp* CrearVentanaImp() = 0;
   virtual ColorImp* CrearColorImp() = 0;
   virtual FuenteImp* CrearFuenteImp() = 0;
// …
// una operación "Crear…" para todos los
recursos
// del sistema de ventanas
};
```

Ahora podemos definir una fábrica concreta para cada sistema de ventanas:

```
class FabricaSistemaDeVentanasPM : public
FabricaSistemaDeVentanas {
   virtual VentanaImp* CrearVentanaImp()
```

```
      { return new VentanaImpPM; }
   // …
};

class   FabricaSistemaDeVentanasX   :   public
FabricaSistemaDeVentanas {
   virtual VentanaImp* CrearVentanaImp()
      { return new VentanaImpX; }
   // …
};
```

El constructor de la clase base Ventana puede usar la interfaz de FabricaSistemaDeVentanas para inicializar el miembro _imp con el objeto VentanaImp apropiado para el sistema de ventanas:

```
Ventana::Ventana() {
   _imp       =       fabricaSistemaDeVentanas-
>CrearVentanaImp();
}
```

La variable fabricaSistemaDeVentanas es una instancia conocida de una subclase FabricaSistemaDeVentanas, similar a la variable fabricaIGU que definía el estándar de interfaz de usuario. La variable fabricaSistemaDeVentanas se puede inicializar del mismo modo.

PATRÓN BRIDGE (PUENTE)

La clase VentanaImp define una interfaz con los servicios comunes de los sistemas de ventanas, pero su diseño viene dado por otras restricciones que las de la interfaz de Ventana. Los programadores de aplicaciones no tratarán directamente con la interfaz de VentanaImp, sino sólo con objetos Ventana. Por tanto, la interfaz de VentanaImp no tiene por qué coincidir con la visión del mundo que tiene el programador de la aplicación, como fue nuestra preocupación al diseñar la interfaz y la jerarquía de clases de Ventana. La interfaz de VentanaImp puede reflejar mejor lo que proporcionan realmente los distintos sistemas de ventanas, con todas sus imperfecciones. Puede aproximarse a uno cualquiera de

los enfoques de intersección o de unión de funcionalidad, el que mejor se adapte al sistema de ventanas de destino.

Lo importante aquí es darse cuenta de que la interfaz de Ventana está dirigida al programador de aplicaciones, mientras que VentanaImp está dirigida a los sistemas de ventanas. Separar la funcionalidad de las ventanas en las jerarquías de Ventana y VentanaImp nos permite implementar y especializar estas interfaces de forma independiente. Los objetos de estas jerarquías cooperan entre sí para que Lexi funcione en varios sistemas de ventanas sin ninguna modificación.

La relación entre Ventana y VentanaImp es un ejemplo del patrón Bridge. El propósito de este patrón es permitir separar jerarquías de clases para que trabajen juntas incluso cuando ambas evolucionan de forma independiente. Nuestro criterio de diseño nos lleva a crear dos jerarquías de clases separadas, una que representa la noción lógica de ventana, y otra para representar diferentes implementaciones de ventanas. El patrón Bridge nos permite mantener y mejorar nuestras abstracciones de ventanas sin tocar el código dependiente del sistema de ventanas, y a la inversa.

2.7. OPERACIONES DE USUARIO

Parte de la funcionalidad de Lexi está disponible en la representación WYSIWYG del documento. El usuario introduce y borra texto, mueve el punto de inserción y selecciona partes del texto mediante el ratón o tecleando directamente sobre el documento. A otras funciones se accede de manera indirecta a través de operaciones de usuario de los mentís desplegables, botones y atajos de teclado de Lexi. La funcionalidad incluye operaciones para

- crear un nuevo documento,
- abrir, guardar e imprimir un documento existente,
- cortar y pegar el texto seleccionado,
- cambiar la fuente y el estilo del texto seleccionado,
- cambiar el formateado del texto, como su alineación y

justificado,
- salir de la aplicación,
- etcétera, etcétera.

Lexi proporciona diferentes interfaces de usuario para llevar a cabo estas operaciones, pero no queremos asociar una determinada operación de usuario con una interfaz en particular, ya que podemos querer tener varias interfaces de usuario para la misma operación (por ejemplo, podemos pasar la página usando un botón o una operación de menú). También podemos querer cambiar la interfaz en un futuro.

Además, estas operaciones están implementadas en muchas clases distintas. Como programadores, queremos acceder a su funcionalidad sin crear muchas dependencias entre las clases de implementación y de usuario. Si no, acabaríamos por tener una implementación fuertemente acoplada, que sería más difícil de comprender, ampliar y mantener.

Para complicar más las cosas, queremos que Lexi soporte operaciones de deshacer y repetir[22] para la mayoría de sus funciones, *pero no para todas*. En concreto, queremos ser capaces de deshacer operaciones que modifican el documento, como eliminar, con las que el usuario puede perder muchos datos sin darse cuenta. Sin embargo, no deberíamos intentar deshacer operaciones como guardar un dibujo o salir de la aplicación. Estas operaciones no deberían tener efecto en el proceso de deshacer. Tampoco queremos un límite arbitrario en el número de niveles de deshacer y repetir.

Está claro que permitir operaciones de usuario es fundamental en la aplicación. El objetivo es conseguir un mecanismo simple y extensible que satisfaga todas estas necesidades.

ENCAPSULAR UNA PETICIÓN

Desde nuestra perspectiva de diseñadores, un menú desplegable no es más que otro tipo de glifo que contiene otros glifos. Lo que distingue a los menús desplegables de otros glifos que tienen hijos es que la mayoría de los glifos de los menús hacen alguna operación en respuesta a un clic del ratón.

Supongamos que estos glifos son instancias de una subclase de

Glifo llamada ElementoDeMenu y que éstos hacen su trabajo en respuesta a una petición de un cliente[23]. Llevar a cabo la petición puede implicar una operación en un objeto, o muchas operaciones sobre muchos objetos, o algo entre medias.

Podríamos definir una subclase de ElementoDeMenu para cada operación de usuario y luego codificar cada subclase para que realizase la petición. Pero eso no es lo correcto: no necesitamos una subclase de ElementoDeMenu para cada petición, de la misma manera que no necesitamos una subclase para cada cadena de texto de un menú desplegable. Es más, este enfoque acopla la petición a una interfaz de usuario determinada, haciendo que sea difícil satisfacer la petición a través de otro elemento de la interfaz.

Para ilustrar esto, supongamos que pudiéramos avanzar a la última página del documento *tanto* con un elemento de menú *como* pulsando en un icono de la interfaz de Lexi (que puede ser más conveniente para documentos cortos). Si asociamos la petición con un ElementoDeMenu a través de la herencia, tendremos que hacer lo mismo para el icono y cualquier otro tipo de útil que pueda responder a dicha petición. Esto puede dar lugar a un número de clases cercano al producto del número de tipos de útiles por el número de peticiones.

Lo que falta es un mecanismo que nos permita parametrizar los elementos de menú con la petición que deben llevar a cabo. De esa forma evitaríamos la proliferación de subclases y permitiríamos más flexibilidad en tiempo de ejecución. Podríamos parametrizar ElementoDeMenu con la función a llamar, pero eso no sería una buena solución, por al menos tres razones:

1. No soluciona el problema de deshacer/repetir.
2. Es difícil asociar el estado a una función. Por ejemplo, una función que cambia la fuente necesita saber *qué* fuente cambiar.
3. Las funciones son difíciles de extender y es difícil reutilizar partes de ellas.

Estas razones sugieren que deberíamos parametrizar los elementos de menú con un *objeto*, y no con una función. De ese modo podríamos usar la herencia para ampliar y reutilizar la

implementación de la petición, y también tendríamos un lugar donde almacenar el estado e implementar la funcionalidad de deshacer/repetir. Éste es otro ejemplo de encapsulación del concepto que varía, en este caso una petición. Encapsularemos cada petición en un objeto orden[24].

CLASE ORDEN Y SUS SUBCLASES

En primer lugar definiremos una clase abstracta Orden que proporciona una interfaz para emitir una petición. La interfaz básica consiste en una única operación abstracta denominada "Ejecutar". Las subclases de Orden implementan Ejecutar de distintas formas para satisfacer diferentes peticiones. Algunas subclases pueden delegar parte del trabajo, o todo, en otros objetos. Otras subclases pueden ser capaces de responder a la petición por sí solas (*véase* la Figura 2.11). No obstante, para el solicitante un objeto Orden no es más que eso, un objeto Orden —todos son tratados por igual—.

Ahora ElementoDeMenu puede almacenar un objeto Orden que satisface una petición (Figura 2.12). Le damos a cada objeto elemento de menú una instancia de la subclase de Orden que resulta

Figura 2.11: Parte de la jerarquía de clases de Orden

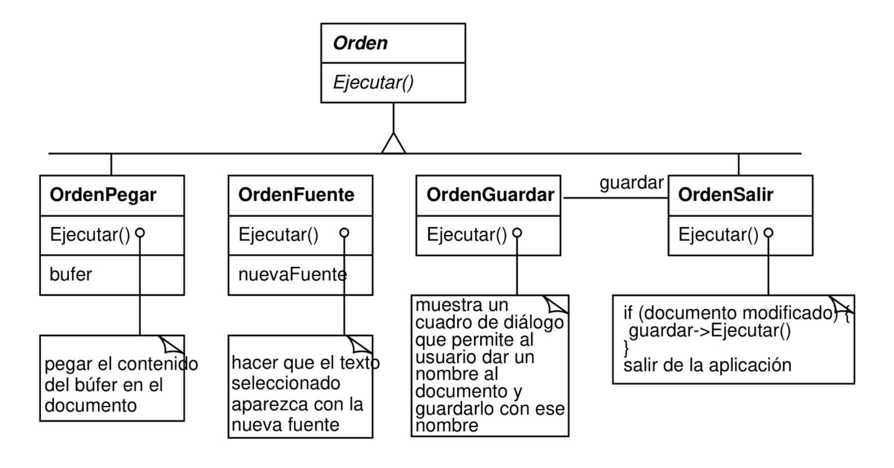
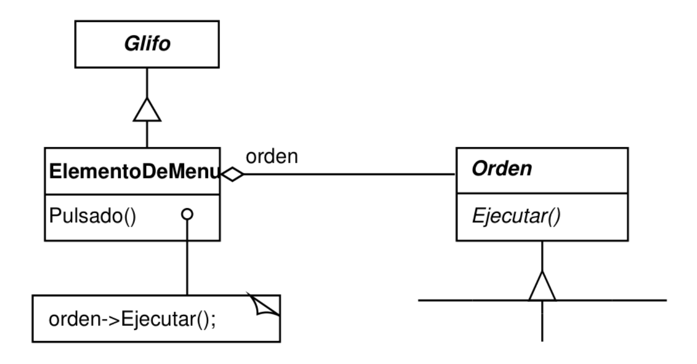


Figure 2.12: Relación entre ElementoDeMenu y Orden



apropiada para ese elemento, de la misma manera que especificamos el texto que aparecerá en él. Cuando el usuario selecciona un elemento de menú en concreto, el correspondiente ElementoDeMenu simplemente llama a Ejecutar sobre su objeto Orden para realizar la petición. Nótese que los botones y otros útiles pueden usar objetos Orden igual que lo hacen los elementos de menú.

CAPACIDAD DE DESHACER [25]

Es importante que las aplicaciones interactivas cuenten con la capacidad de deshacer/repetir. Para deshacer y repetir órdenes, añadiremos una operación Deshacer a la interfaz de Orden, que anula los efectos de la operación Ejecutar precedente, usando la información que ésta haya guardado. Por ejemplo, en el caso de una OrdenFuente, la operación Ejecutar debería guardar la parte del texto a la que afecta el cambio de fuente junto con la fuente o fuentes originales. La operación Deshacer de OrdenFuente devolvería el texto a su fuente original.

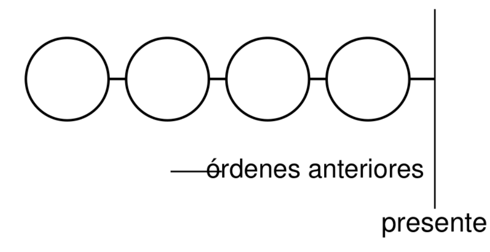
A veces la capacidad de deshacer tiene que ser establecida en tiempo de ejecución. Una petición para cambiar la fuente de una selección no hace nada si el texto ya aparece en esa fuente. Supongamos que el usuario selecciona parte del texto y realiza un cambio de fuente que no tiene ningún efecto. ¿Cuál debería ser el resultado de la siguiente petición deshacer? ¿Un cambio sin sentido debería causar que la petición deshacer hiciera algo igualmente sin sentido? Probablemente no. Si el usuario repite ese cambio de fuente varias veces, no debería tener que realizar exactamente el mismo número de operaciones deshacer para regresar a la última

operación con sentido. Si el resultado final de ejecutar una orden fue nulo, entonces no hay necesidad de la correspondiente operación deshacer.

Así que para determinar si una orden se puede deshacer, añadiremos una operación abstracta Reversible a la interfaz de Orden. Reversible devuelve un valor lógico, y las subclases de Orden pueden redefinir esta operación para devolver verdadero o falso basándose en criterios de tiempo de ejecución.

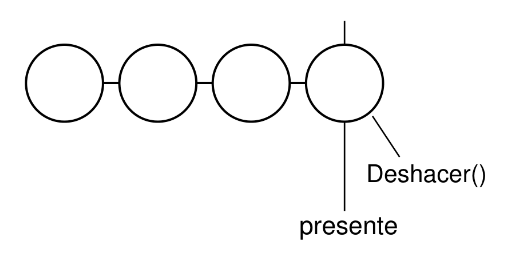
HISTORIAL DE ÓRDENES

El paso final para soportar niveles arbitrarios de deshacer y repetir es definir un historial de órdenes, es decir, una lista de las órdenes ejecutadas (y las anuladas). Conceptualmente, el historial de órdenes se parece a esto:

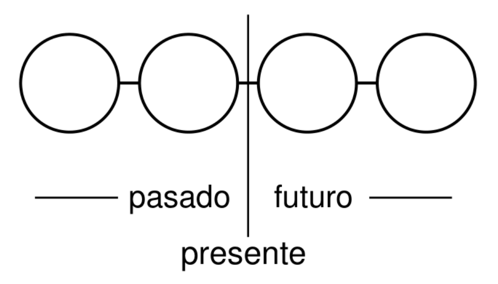


Cada círculo representa un objeto Orden. En este caso el usuario ha ejecutado cuatro órdenes. La de más a la izquierda fue la primera que se ejecutó, seguida por la segunda de más a la izquierda y así sucesivamente hasta la orden más recientemente ejecutada, que es la de más a la derecha. La línea etiquetada como “presente” lleva la cuenta de cuál fue la última orden ejecutada (o anulada).

Para deshacer la última orden, simplemente llamaremos a Deshacer sobre la orden más reciente:

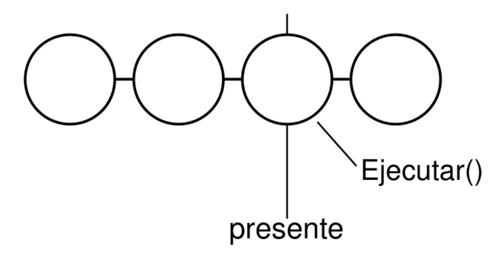


Después de la operación de deshacer, movemos la línea “presente” una orden hacia la izquierda. Si el usuario elige de nuevo deshacer, se anulará la siguiente orden más recientemente ejecutada de la misma manera que se hizo con ésta, y nos encontraremos en el estado que se muestra a continuación:

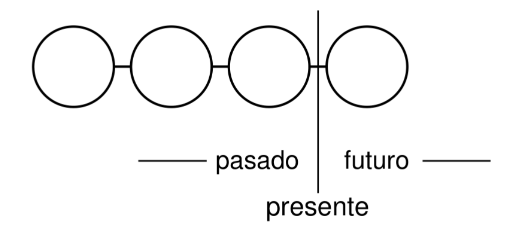


Puede observarse cómo simplemente repitiendo este procedimiento obtenemos múltiples niveles de operaciones de deshacer. El número de estos niveles sólo está limitado por la longitud del historial de órdenes.

Para repetir una orden que se acaba de deshacer, hacemos lo mismo pero a la inversa. Las órdenes a la derecha de la línea del presente son las que se pueden deshacer en un futuro. Para repetir la última orden anulada, llamamos a Ejecutar en la orden situada a la derecha de la línea actual:



A continuación movemos la línea del presente de manera que la siguiente operación de repetir llamará a repetir sobre la siguiente orden en el futuro.



Por supuesto, si la siguiente operación no es otro repetir, sino un deshacer, entonces se anulará la orden a la izquierda de la línea "presente". De esta manera el usuario puede ir hacia delante y hacia atrás en el tiempo tanto como sea necesario para recuperarse de los errores.

PATRÓN COMMAND (ORDEN)

Las órdenes de Lexi son una aplicación del patrón Command (215), el cual describe cómo encapsular una petición. El patrón Command prescribe una interfaz uniforme para emitir peticiones que permite configurar los clientes para que traten diferentes peticiones. La interfaz oculta a los clientes la implementación de las peticiones. Una orden puede delegar toda la implementación de la petición, parte de ella o nada en otros objetos. Esto es perfecto para aplicaciones como Lexi, que deben proporcionar un acceso centralizado a la funcionalidad que se encuentra desperdigada por toda la aplicación. El patrón también trata los mecanismos de deshacer y repetir incorporados en la interfaz básica de Orden.

2.8. REVISIÓN ORTOGRÁFICA E INSERCIÓN DE GUIONES[26]

El último problema de diseño tiene que ver con el análisis del texto, en concreto con la revisión ortográfica y la introducción de puntos de inserción de guiones donde sea necesario para lograr un buen formateado.

Las restricciones son similares a las que tuvimos en el problema de diseño del formateado, en la Sección 2.3. Al igual que con las

estrategias de división de líneas, hay más de una manera de revisar la ortografía y de calcular los puntos donde se pueden insertar guiones de separación. Por tanto, aquí también queremos permitir múltiples algoritmos. Un conjunto de algoritmos puede proporcionar una elección de compromiso entre espacio/tiempo/calidad.

También queremos evitar implementar esta funcionalidad en la estructura del documento. Este objetivo es incluso más importante aquí que en el caso del formateado, ya que la revisión ortográfica y la inserción de guiones son sólo dos de todos los tipos potenciales de análisis que podemos querer que permita Lexi. Sin duda, querremos ampliar las capacidades analíticas de Lexi con el tiempo. Podemos añadir búsquedas, herramientas de contar palabras, una utilidad de cálculo para sumar los valores de una tabla, revisión gramatical, etcétera. Pero no queremos cambiar la clase Glifo y todas sus subclases cada vez que introducimos nuevas funcionalidades de este tipo.

Este rompecabezas lo forman en realidad dos piezas: (1) acceder a la información que va a ser analizada, que se encuentra repartida entre los glifos de la estructura del documento, y (2) hacer el análisis. Veamos ambas piezas por separado.

ACCEDER A INFORMACIÓN DISPERSA

Hay muchos tipos de análisis que requieren examinar el texto carácter a carácter. El texto a analizar está repartido en una estructura jerárquica de objetos glifo. Para examinar el texto en dicha estructura necesitamos un mecanismo de acceso que conozca las estructuras de datos en las que se almacenan los objetos. Algunos glifos pueden almacenar sus hijos en listas enlazadas, otros tal vez usen arrays, y otros pueden usar estructuras de datos más extrañas. Nuestro mecanismo de acceso debe ser capaz de tratar con todas estas posibilidades.

Una complicación añadida es que los diferentes análisis acceden a la información de diferentes formas. La mayoría de los análisis recorrerán el texto de principio a fin, pero algunos harán lo contrario —una búsqueda hacia atrás, por ejemplo, necesita examinar el texto hacia atrás en vez de hacia delante—. Otras operaciones, como la evaluación de expresiones algebraicas,

podrían necesitar un recorrido en-orden.

Por tanto, nuestro mecanismo de acceso se tiene que acomodar a diferentes estructuras de datos, y debemos permitir diferentes tipos de recorridos, como pre-orden, post-orden y en-orden.

ENCAPSULAR EL ACCESO Y EL RECORRIDO

Por ahora nuestra interfaz de glifo usa un índice entero para que los clientes se refirieran a sus hijos. Si bien esto puede ser razonable para las clases de glifo que almacenan sus hijos en un array, puede ser ineficiente para glifos que utilizan una lista enlazada. Un papel importante de la abstracción de glifo es ocultar la estructura de datos en la que se almacenan los hijos. De ese modo podemos cambiar la estructura de datos de un glifo sin que esto afecte a otras clases.

Por tanto, sólo el glifo puede saber la estructura de datos que usa. Como corolario, la interfaz de glifo no debería orientarse hacia una estructura de datos u otra; no debería adecuarse mejor a arrays que a listas enlazadas, por ejemplo.

Podemos solucionar este problema y permitir diferentes tipos de recorridos al mismo tiempo. Podemos poner diferentes capacidades de acceso y recorrido directamente en las clases glifo y proporcionar un modo de elegir entre ellas, tal vez mediante una constante enumerada como parámetro. Las clases se pasarían este parámetro de unas a otras durante un recorrido para garantizar que están haciendo el mismo tipo de recorrido. También deben pasar cualquier información acumulada durante el recorrido.

Podríamos añadir las siguientes operaciones abstractas a la interfaz de Glifo para permitir este enfoque:

```
void Primero(Recorrido tipo)
void Siguiente()
bool HaTerminado()
Glifo* ObtenerActual()
void Insertar(Glifo*)
```

Las operaciones Primero, Siguiente y HaTerminado controlan el recorrido. Primero lo inicializa. Recibe el tipo de recorrido como un

parámetro de tipo Recorrido, una constante enumerada que tiene valores tales como HIJOS (para recorrer sólo los hijos directos del glifo), PREORDEN (para recorrer la estructura completa en preorden), POSTORDEN y ENORDEN. Siguiente avanza hasta el siguiente glifo del recorrido, y HaTerminado informa si el recorrido ha terminado o no. ObtenerActual sustituye a la operación Hijo, y accede al glifo actual del recorrido. Insertar sustituye la operación antigua, insertando el glifo dado en la posición actual.

Un análisis usaría el siguiente código C++ para hacer un recorrido preorden de una estructura de glifos que tiene como raíz g:

```
Glifo* g;

for (g->Primero(PREORDEN); !g->HaTerminado();
g->Siguiente()) {
   Glifo* actual = g->ObtenerActual();

   // hace algún análisis
}
```

Nótese cómo ha desaparecido el índice entero de la interfaz de glifo. Ya no hay nada que predisponga la interfaz hacia un tipo de colección u otra. También les ahorramos a los clientes tener que implementar ellos mismos algunos tipos de recorridos habituales.

Pero este enfoque sigue teniendo problemas. Por un lado, no pueden permitir nuevos recorridos sin ampliar el conjunto de valores enumerados o sin añadir nuevas operaciones. Supongamos que queremos una variación de un recorrido preorden que se salte automáticamente glifos no textuales. Tendríamos que cambiar la enumeración Recorrido para incluir algo como PREORDEN_TEXTUAL.

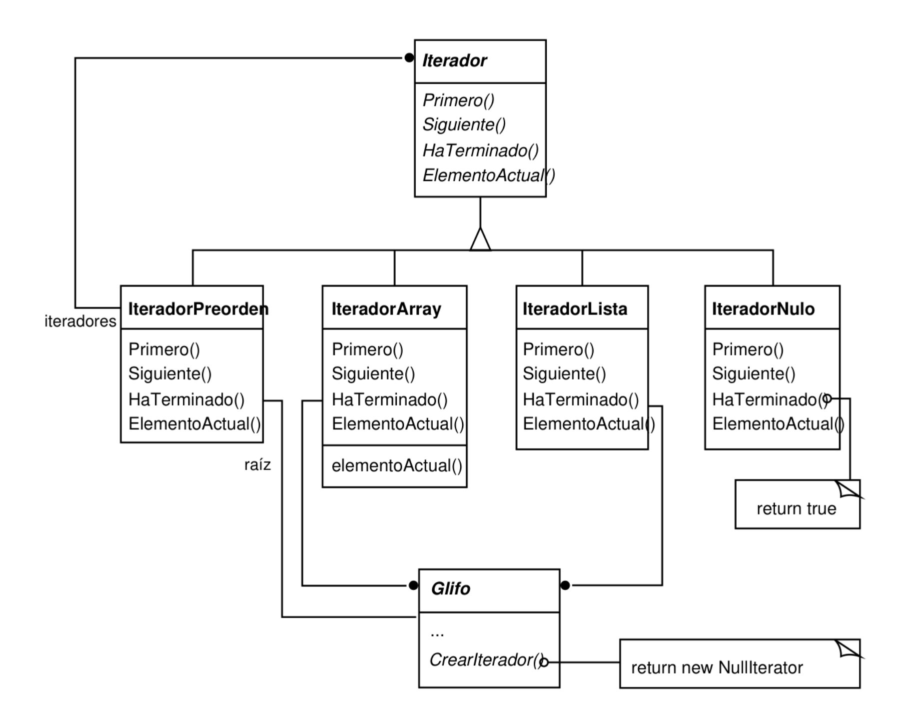
Nos gustaría no tener que cambiar las declaraciones existentes. Poner todo el mecanismo de recorrido en la jerarquía de la clase Glifo haría que fuese difícil modificarlo o ampliarlo sin cambiar a su vez muchas otras clases. También sería difícil reutilizar el mecanismo para recorrer otros tipos de estructuras de objetos. Y no podemos hacer más de un recorrido a la vez sobre la misma estructura.

Una solución mejor es, de nuevo, encapsular el concepto que varía, en este caso los mecanismos de acceso y recorrido. Podemos introducir una clase de objetos llamados iteradores, cuyo único propósito es definir diferentes tipos de tales mecanismos. Podemos usar la herencia para poder acceder a estructuras de datos diferentes de una manera uniforme, así como permitir nuevos tipos de recorridos. Y no tendremos que cambiar las interfaces de los glifos o tocar sus implementaciones para hacerlo.

CLASE ITERADOR Y SUS SUBCLASES

Usaremos una clase abstracta llamada Iterador para definir una interfaz general que permita recorrer una estructura de objetos y acceder a cada uno de sus elementos. Las subclases concretas como Iterador Array o IteradorLista implementan la interfaz para proporcionar acceso a arrays y listas, mientras que IteradorPreorden. IteradorPostorden y similares implementan diferentes recorridos sobre determinadas estructuras. Cada subclase de Iterador tiene una referencia a la estructura que recorre. Las instancias de estas subclases se inicializan con dicha referencia cuando se crean. La Figura 2.13 muestra la clase Iterador junto con varias subclases. Nótese que hemos añadido una operación abstracta CrearIterador a la interfaz de la clase Glifo para dar cabida a los iteradores.

Figura 2.13: Clase Iterador y sus subclases



La interfaz Iterador provee las operaciones Primero, Siguiente y HaTerminado para controlar el recorrido. La clase IteradorLista implementa Primero para que apunte al primer elemento de la lista, y Siguiente hace avanzar al iterador hasta el siguiente elemento de la lista. HaTerminado devuelve si el puntero de la lista apunta más allá del último elemento de la lista. ElementoActual desreferencia el iterador para devolver el glifo al que apunta. Una clase IteradorArray haría cosas similares pero sobre un array de glifos.

Ahora podemos acceder a los hijos de una estructura de glifos sin conocer su representación:

```
Glifo* g;
Iterador<Glifo*>* i = g->CrearIterador();

for  (i->Primero();  !i->HaTerminado();  i-
>Siguiente()) {
  Glifo* hijo = i->ElementoActual();

  // hace algo con el hijo actual
}
```

CrearIterador devuelve, de manera predeterminada, una instancia de IteradorNulo. Un IteradorNulo es un iterador degenerado para glifos que no tienen ningún hijo, es decir, para los glifos hoja. La operación HaTerminado de un IteradorNulo siempre devuelve verdadero.

Una subclase de glifo que tiene hijos redefinirá CrearIterador para devolver una instancia de una subclase de Iterador diferente. La subclase concreta depende de la estructura donde se guardan los hijos. Si la subclase de Glifo, Fila, guarda sus hijos en una lista _hijos, entonces su operación CrearIterador se parecerá a esto:

```
Iterador<Glifo*>* Fila::CrearIterador () {
   return new IteradorLista<Glifo*>(_hijos);
}
```

Los iteradores para recorridos preorden y enorden se implementan en términos de iteradores específicos de glifos. Los iteradores para estos recorridos son proporcionados por el glifo raíz de la estructura que recorren. Llaman a CrearIterador sobre los glifos de la estructura y usan una pila para guardar los iteradores resultantes.

Por ejemplo, la clase IteradorPreorden obtiene el iterador del glifo raíz, lo inicializa para que apunte a su primer elemento y luego lo mete en la pila:

```
void IteradorPreorden::Primero () {
   Iterador<Glifo*>*      i      =      _raiz-
>CrearIterador();

   if (i) {
      i->Primero();
      _iteradores.BorrarTodos();
      _iteradores.Meter(i);
   }
}
```

ElementoActual simplemente llamaría a ElementoActual del iterador de la cima de la pila:

```
Glifo*   IteradorPreorden::ElementoActual   ()
const {
   return
      _iteradores.Tamano() > 0 ?
      _iteradores.Cima()->ElementoActual()    :
0;
}
```

La operación Siguiente devuelve el iterador de la cima de la pila y le pide a su elemento actual que cree un iterador, en un intento de descender en la estructura de glifos tanto como sea posible (después de todo, en esto consiste un recorrido en preorden). Siguiente hace que el nuevo iterador apunte al primer elemento a recorrer y lo mete en la pila. Después, Siguiente comprueba el último iterador; si su operación HaTerminado devuelve verdadero significa que hemos terminado de recorrer el subárbol (o la hoja) actual. En ese caso, Siguiente saca el iterador de la cima de la pila y repite este proceso hasta que encuentra el siguiente recorrido incompleto, si es que hay alguno; si no, es que hemos terminado de recorrer la estructura.

```
void IteradorPreorden::Siguiente () {
   Iterator<Glifo*>* i =
      _iteradores.Cima()->ElementoActual()-
>CrearIterador();

   i->Primero();
   _iteradores.Meter(i);

   while (_iteradores.Tamano() > 0 &&
           _iteradores.Cima()->HaTerminado())
{
      delete _iteradores.Sacar();
      _iteradores.Cima()->Siguiente();
   }
}
```

Nótese cómo la clase Iterador nos permite añadir nuevos tipos de recorridos sin modificar las clases de glifos, simplemente heredamos de Iterador y añadimos un nuevo recorrido, como hemos hecho con IteradorPreorden. Las subclases de Glifo usan la misma interfaz para dar a los clientes acceso a sus hijos sin revelar la estructura de datos

subyacente que usan para almacenarlos. Como los iteradores guardan su propia copia del estado del recorrido, podemos realizar varios recorridos simultáneamente, incluso sobre la misma estructura. Y aunque los recorridos que hemos hecho en este ejemplo han sido sobre estructuras de glifos, no hay ninguna razón por la que no podamos parametrizar una clase como IteradorPreorden con el tipo del objeto de la estructura. En C++ lo haríamos mediante plantillas. Acto seguido podríamos reutilizar el mecanismo de IteradorPreorden para recorrer otras estructuras.

PATRÓN ITERATOR (ITERADOR)

El patrón Iterator (237) representa estas técnicas para permitir recorrer estructuras de objetos y acceder a sus elementos. No sólo se puede aplicar a estructuras compuestas, sino también a colecciones. Abstrae el algoritmo de recorrido y oculta a los clientes la estructura interna de los objetos que recorre. El patrón Iterator ilustra, una vez más, cómo al encapsular el concepto que varía ganamos en flexibilidad y reutilización. Incluso así, el problema de la iteración es sorprendentemente complejo, y el patrón Iterator tiene muchos más matices y ventajas e inconvenientes que los que hemos considerado aquí.

RECORRIDO Y ACCIONES DEL RECORRIDO

Ahora que tenemos un modo de recorrer la estructura de glifos, necesitamos comprobar la ortografía y la inserción de guiones de separación. Ambos análisis implican acumular información durante el recorrido.

En primer lugar tenemos que decidir dónde poner la responsabilidad del análisis. Podríamos ponerla en las clases Iterador, haciendo así al análisis una parte integral del recorrido. Pero tendremos mucha más flexibilidad y posibilidad de reutilización si diferenciamos entre el recorrido y las acciones a realizar durante dicho recorrido. Eso es debido a que distintos análisis muchas veces requieren el mismo tipo de recorrido. De ahí que podamos reutilizar el mismo conjunto de iteradores para diferentes análisis. Por ejemplo, el recorrido en preorden es común a muchos análisis, incluyendo la revisión ortográfica, la separación

de palabras mediante guiones, la búsqueda hacia delante y el recuento de palabras.

Por tanto habría que separar el análisis y el recorrido. ¿En qué otro lugar podemos poner la responsabilidad del análisis? Sabemos que hay muchos tipos de análisis que podemos querer hacer. Cada análisis hará diferentes cosas en distintos puntos del recorrido. Algunos glifos son más significativos que otros, dependiendo del tipo de análisis. Si estamos comprobando la ortografía o la inserción de guiones, querremos tener en cuenta los glifos de carácter y no los gráficos, como líneas o imágenes de mapas de bits. Si estamos haciendo separaciones de color, nos interesarán los glifos visibles y no los invisibles. Es inevitable que haya análisis diferentes para glifos diferentes.

Por tanto, un determinado análisis debe ser capaz de distinguir diferentes tipos de glifos. Un enfoque obvio es poner la capacidad analítica en las propias clases glifo. Para cada análisis podemos añadir una o más operaciones abstractas a la clase Glifo y tener subclases que las implementen según el papel que cada una desempeñe en el análisis.

Pero el problema de ese enfoque es que tendremos que cambiar cada clase glifo cada vez que añadamos un nuevo tipo de análisis. Podemos paliar este problema en algunos casos: si en el análisis participan unas pocas clases, o si la mayoría de las clases hacen el análisis de la misma manera, podemos proporcionar una implementación predeterminada para la operación abstracta en la clase Glifo. La operación predeterminada cubriría así el caso general. De esa manera limitaríamos los cambios a la clase Glifo y a aquellas subclases que se aparten de la norma general

Si bien una implementación predeterminada reduce el número de cambios, todavía permanece un problema insidioso: la interfaz de Glifo se expande con cada nueva capacidad de análisis. A medida que pase el tiempo las operaciones de análisis comenzarán a oscurecer la interfaz básica de Glifo, haciendo difícil ver que el principal propósito de un glifo es definir una estructura de objetos que tienen apariencia y forma (esa interfaz se pierde entre el ruido).

ENCAPSULAR EL ANÁLISIS

Según todos los indicios, necesitamos encapsular el análisis en un

objeto aparte, de forma similar a como hemos hecho muchas veces hasta ahora. De ese modo pondríamos todos los mecanismos de un determinado análisis en su propia clase. Podríamos usar una instancia de esta clase en conjunción con el iterador apropiado. El iterador "llevaría" la instancia a cada glifo de la estructura. Este objeto podría entonces realizar una porción del análisis en cada punto del recorrido. El analizador acumula información de interés (caracteres en este caso) a medida que avanza el recorrido (*véase* la figura de la página siguiente).

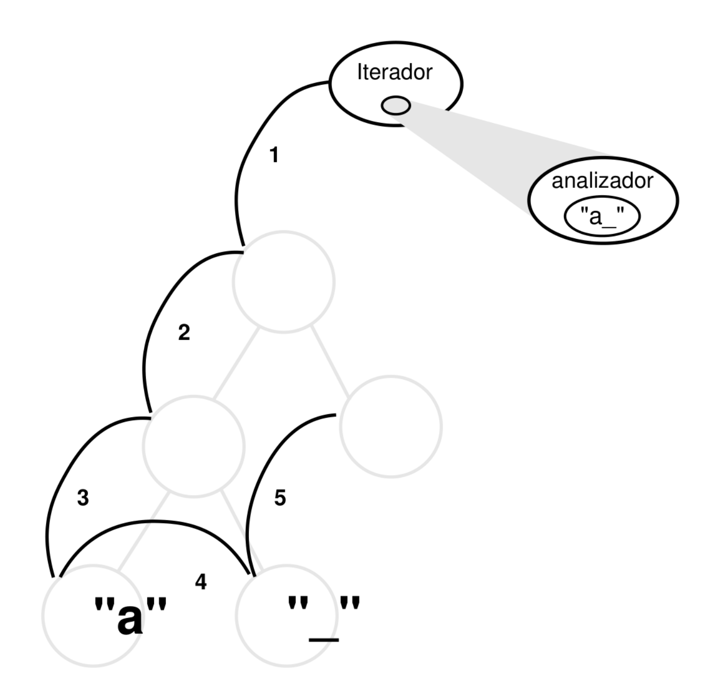
La cuestión fundamental con este enfoque es cómo distingue el objeto de análisis los tipos de glifos sin recurrir a comprobaciones de tipo o conversiones a un tipo inferior (*downcast*). No queremos una clase RevisorOrtografico que incluya (pseudo)código como

```
void   RevisorOrtografico::Revisar   (Glifo*
glifo) {
   Caracter* c;
   Fila* f;
   Imagen* i;

   if (c = dynamic_cast<Caracter*>(glifo)) {
      // analiza el carácter

   } else if (f = dynamic_cast<Fila*>(glifo))
{
      // se prepara para analizar sus hijos

   }       else       if       (i       =
dynamic_cast<Imagen*>(glifo)) {
      // no hace nada
   }
}
```



Este código es bastante feo. Se basa en características bastante esotéricas como los ahormados de tipo seguros. También resulta difícil de ampliar. Tendremos que recordar cambiar el cuerpo de esta función cada vez que cambiemos la jerarquía de la clase Glifo. De hecho, ésta es la clase de código que los lenguajes orientados a objetos trataban de eliminar.

Queremos evitar tal enfoque basado en la fuerza bruta, pero ¿cómo? Pensemos qué sucede cuando añadimos la siguiente operación abstracta a la clase Glifo:

```
void Revisame(RevisorOrtografico&)
```

Definimos Revísame en cada subclase de Glifo como sigue:

```
void
SubclaseGlifo::Revisame(RevisorOrtografico&
revisor) {
   revisor.RevisarSubclaseGlifo(this);
}
```

donde SubclaseGlifo será reemplazada por el nombre de la subclase del glifo. Nótese que cuando se llama a Revisame, se conoce la subclase concreta de Glifo —después de todo, estamos en una de sus operaciones—. A su vez, la interfaz de la clase RevisorOrtografico incluye una operación como RevisarSubclaseGlifo para cada subclase de Glifo [27]:

```
class RevisorOrtografico {
public:
   RevisorOrtografico();

   virtual void RevisarCaracter(Caracter*);
   virtual void RevisarFila(Fila*);
   virtual void RevisarImagen(Imagen*);

   // … etcétera

   Lista<char*>&
ObtenerErroresOrtograficos();

protected:
   virtual bool EstaMalEscrito(const char*);

private:
   char _palabraActual[MAX_TAM_PALABRA];
   Lista<char*> _errores;
};
```

La operación del RevisorOrtografico para glifos de Caracter puede parecerse a esto:

```
void       RevisorOrtografico::RevisarCaracter
(Caracter* c) {
   const       char       c       =       c-
>ObtenerCodigoDeCaracter();

   if (isalpha(c)) {
      //  añade  el  caracter  alfabético  a
_palabraActual

   } else {
      // hemos encontrado un glifo que no es
un caracter
```

```
      if (EstaMalEscrito(_palabraActual)) {
         // añade la _palabraActual a _errores
         _errores.Insertar(strdup(_palabraActual));
      }

      _palabraActual[0] = '\0';
         //  inicializa  _palabraActual  para
comprobar
         //la siguiente palabra
   }
}
```

Nótese que hemos definido una operación especial ObtenerCodigoDeCaracter en la clase Carácter. El revisor gramatical puede tratar con operaciones específicas de las subclases sin depender de comprobaciones de tipo o ahormados —nos permite tratar a los objetos de forma especial—.

RevisarCaracter acumula caracteres alfabéticos en el búfer _palabraActual[28]. Cuando encuentra un carácter no alfabético, como un guión de subrayado, hace uso de la operación EstaMalEscrito para comprobar la ortografía de la palabra que está en _palabraActual. Si la palabra está mal escrita, RevisarCaracter la añade a la lista de palabras mal escritas. A continuación debe vaciar el búfer _palabraActual para dejarlo listo para la siguiente palabra. Cuando termina el recorrido, se puede obtener la lista de palabras mal escritas con la operación ObtenerErroresOrtograficos.

Ahora podemos recorrer la estructura, llamando a Revisame sobre cada glifo, con el revisor gramatical como argumento. Esto permite al RevisorOrtografico identificar a cada glifo.

```
RevisorOrtografico revisorOrtografico;
Composicion* c;

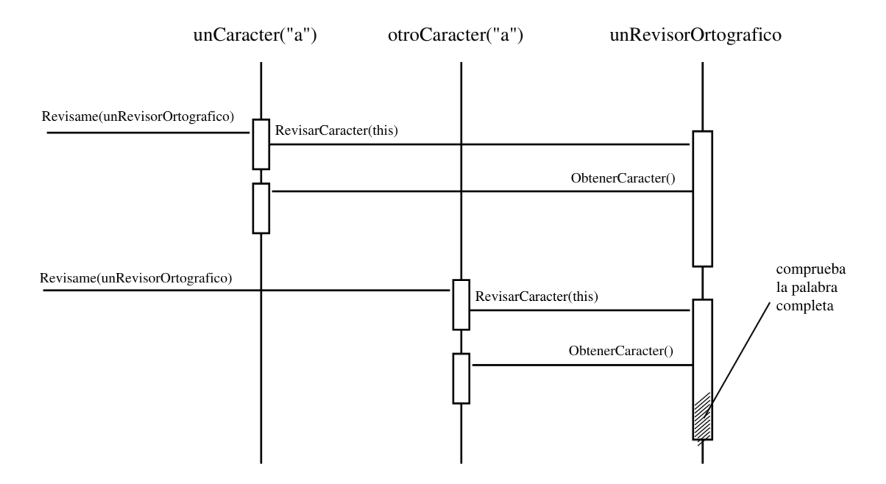
// …

Glifo* g;
IteradorPreorden i(c);

for     (i.Primero();     !i.HaTerminado();
i.Siguiente()) {
   g = i.ElementoActual();
   g->Revisame(revisorOrtografico);
```

}

El siguiente diagrama de interacción muestra cómo trabajan juntos los glifos de Carácter y el objeto RevisorOrtografico:



Este enfoque nos sirve para encontrar errores gramaticales, pero ¿cómo nos ayuda a permitir múltiples tipos de análisis? Parece como si tuviéramos que añadir una operación como Revísame (RevisorOrtografico&) a Glifo y sus subclases cada vez que añadamos un nuevo tipo de análisis. Esto es cierto si insistimos en una clase *independiente* para cada análisis. Pero no hay ningún motivo que nos impida dar la misma interfaz a *todas* las clases de análisis. Al hacerlo así podemos usarlas polimórficamente. Eso significa que podemos reemplazar operaciones específicas de cada análisis como Revísame(RevisorOrtografico&) con una operación independiente del análisis que tome un parámetro más general.

CLASE VISITANTE Y SUS SUBCLASES

Usaremos el término visitante para referirnos generalmente a clases de objetos que "visitan" otros objetos durante un recorrido y hacen algo apropiado[29]. En este caso podemos definir una clase Visitante con una interfaz abstracta para visitar glifos en una estructura.

```
class Visitante {
public:
  virtual  void  VisitarCaracter(Caracter*)  {
}
  virtual void VisitarFila(Fila*) { }
  virtual void VisitarImagen(Imagen*) { }

  // … etcétera
};
```

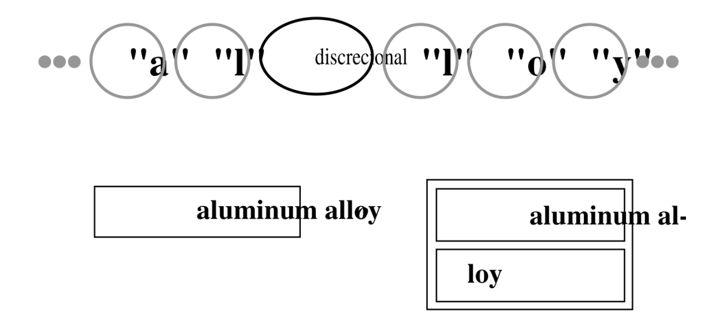
Las subclases concretas de Visitante realizan distintos análisis. Por ejemplo, podríamos tener una subclase VisitanteRevisionOrtografica para comprobar la ortografía, y una subclase VisitanteSeparacionGuiones para la división de las palabras con guiones al final de la línea. VisitanteRevisionOrtografica se implementaría exactamente como implementamos antes el RevisorOrtografico, salvo los nombres de la operación, que reflejarían la interfaz más general de Visitante. Por ejemplo, RevisarCaracter se llamaría VisitarCaracter.

Como Revísame no resulta apropiado para los visitantes que no comprueban nada, le daremos un nombre más general: Aceptar. Su argumento también tiene que cambiar para recibir un Visitantes, reflejando el hecho de que puede aceptar cualquier visitante. Ahora, para añadir un nuevo análisis basta con definir una nueva subclase de Visitante —no tenemos que tocar ninguna de las clases glifo—. Añadiendo la citada operación a Glifo y sus subclases permitimos cualquier análisis futuro.

Ya hemos visto cómo funciona la revisión ortográfica. Usaremos un enfoque similar en VisitanteSeparacionGuiones para acumular texto. Pero una vez que la operación VisitarCaracter de VisitanteSeparacionGuiones ha ensamblado una palabra completa, funciona algo diferente. En vez de comprobar la ortografía de la palabra, aplica un algoritmo de separación de sílabas para determinar los puntos potenciales de inserción de guiones en la palabra, si es que hay alguno. Entonces, en cada punto de inserción de guiones, inserta un glifo **discrecional** en la composición. Los glifos discrecionales son instancias de Discrecional, una subclase de Glifo.

Un glifo discrecional tiene dos apariencias posibles, dependiendo

de si es o no el último carácter de una línea. Si es el último carácter, entonces se representa como un guión; si no, no tiene ninguna representación. El glifo discrecional comprueba su padre (un objeto Fila) para ver si es el último hijo. Hace esta operación cada vez que es llamado para que se dibuje o calcule sus límites. La estrategia de formateado trata a los glifos discrecionales como a los espacios en blanco, haciéndolos candidatos para terminar una línea. El siguiente diagrama muestra cómo puede aparecer un glifo discrecional insertado.



PATRÓN VISITOR (VISITANTE)

Lo que se ha descrito hasta aquí es una aplicación del patrón Visitor (305). La clase Visitante que se describió anteriormente y sus subclases son los participantes clave del patrón. El patrón Visitor representa la técnica que hemos usado para permitir un número indefinido de análisis de estructuras de glifos sin tener que cambiar las propias clases de los distintos glifos. Otra característica interesante de los visitantes es que se pueden aplicar no sólo a elementos compuestos como nuestras estructuras de glifos, sino a cualquier estructura de objetos. Eso incluye conjuntos, listas e incluso grafos dirigidos acíclicos. Es más, las clases que puede visitar un visitante no necesitan estar relacionadas unas con otras por medio de una clase padre común. Esto significa que los visitantes pueden funcionar en varias jerarquías de clases.

Una cuestión importante que debemos preguntamos antes de aplicar el patrón Visitor es: ¿qué jerarquías de clases cambian más frecuentemente? El patrón resulta más apropiado cuando queremos ser capaces de realizar varias operaciones diferentes sobre unos objetos que tienen una estructura de clases estable. Añadir un

nuevo tipo de visitante no requiere ningún cambio en la estructura de clases, lo que es especialmente importante cuando la estructura de clases es grande. Pero cada vez que se añade una subclase a la estructura, también habrá que actualizar todas nuestras interfaces de visitantes para incluir una operación Visitar… para cada subclase. En nuestro ejemplo, eso significa que añadir una nueva subclase de Glifo llamada Foo requerirá cambiar Visitante y todas sus subclases para que incluyan una operación VisitarFoo. Pero, dadas las restricciones de nuestro diseño, es mucho más probable que añadamos nuevos tipos de análisis a Lexi que un nuevo tipo de Glifo. De manera que el patrón Visitor resulta apropiado para nuestras necesidades.

2.9. RESUMEN

Hemos aplicado ocho patrones diferentes al diseño de Lexi:

1. Composite (151) para representar la estructura física del documento,
2. Strategy (289) para permitir diferentes tipos de algoritmos de formateado.
3. Decorator (161) para adornar la interfaz de usuario,
4. Abstract Factory (79) para permitir múltiples estándares de interfaz de usuario,
5. Bridge (141) para permitir diferentes plataformas de ventanas,
6. Command (215) para deshacer operaciones de usuario,
7. Iterator (237) para recorrer estructuras de objetos y acceder a sus elementos, y
8. Visitor (305) para permitir un número indeterminado de capacidades de análisis sin complicar la implementación de la estructura del documento.

Ninguno de estos problemas de diseño se reduce a aplicaciones de edición de documentos como Lexi. De hecho, la mayoría de aplicaciones no triviales tendrán ocasión de usar muchos de estos

patrones, aunque tal vez para hacer cosas diferentes. Una aplicación de análisis financiero podría usar el patrón Composite para definir carteras de inversiones formadas por subcarteras y cuentas de diferentes tipos. Un compilador podría usar el patrón Strategy para permitir diferentes esquemas de asignación de registros para diferentes máquinas de destino. Las aplicaciones con una interfaz gráfica probablemente aplicarán al menos los patrones Decorator y Command tal y como hemos hecho aquí.

Aunque hemos cubierto varios problemas importantes del diseño de Lexi, hay muchos otros que no hemos discutido. Este libro describe más de los ocho patrones que hemos usado aquí. Así que, a medida que estudie el resto de patrones, piense cómo podría usar cada uno de ellos en Lexi. O, mejor aún, ¡piense cómo utilizarlos en sus propios diseños!

Catálogo de Patrones de Diseño



CONTENIDO DEL CAPÍTULO

Los patrones de diseño de creación abstraen el proceso de creación de instancias. Ayudan a hacer a un sistema independiente de cómo se crean, se componen y se representan sus objetos. Un patrón de creación de clases usa la herencia para cambiar la clase de la instancia a crear, mientras que un patrón de creación de objetos delega la creación de la instancia en otro objeto.

Los patrones de creación se hacen más importantes a medida que los sistemas evolucionan para depender más de la composición de objetos que de la herencia de clases. Cuando esto sucede, se pasa de codificar una serie de comportamientos fijos a definir un conjunto más pequeño de comportamientos fundamentales que pueden componerse con otros más complejos. Así, para crear objetos con un determinado comportamiento es necesario algo más que simplemente crear una instancia de una clase.

Hay dos temas recurrentes en estos patrones. En primer lugar, todos ellos encapsulan el conocimiento sobre las clases concretas que usa el sistema. Segundo, todos ocultan cómo se crean y se asocian las instancias de estas clases. Todo lo que el sistema como tal conoce acerca de los objetos son sus interfaces, tal y como las definen sus clases abstractas. Por tanto, los patrones de creación dan mucha flexibilidad a qué es lo que se crea, quién lo crea y cuándo. Permiten configurar un sistema con objetos “producto” que varían mucho en estructura y funcionalidad. La configuración puede ser estática (esto es, especificada en tiempo de compilación) o dinámica (en tiempo de ejecución).

A veces, los patrones de creación son rivales entre sí. Por ejemplo, hay casos en los que tanto el patrón Prototype (109) como el Abstract Factory (79) podrían usarse provechosamente. Otras veces son complementarios: el patrón Builder (97) puede usar uno de los otros patrones de creación para implementar qué componentes debe construir: el patrón Prototype (109) puede usar el Singleton (119) en su implementación.

Dado que los patrones de creación están estrechamente relacionados, estudiaremos los cinco juntos para resaltar sus similitudes y diferencias. También usaremos un ejemplo común —construir un laberinto para un juego de computadora— para ilustrar sus implementaciones. El laberinto y el juego cambiarán ligeramente de un patrón a otro. A veces el juego consistirá simplemente en encontrar la salida del laberinto: en ese caso el jugador probablemente sólo tendrá una visión local del laberinto. Otras veces el laberinto contendrá problemas que resolver y peligros que superar, y estos juegos pueden proporcionar un plano de la parte del laberinto que ha sido explorada.

Omitiremos muchos detalles sobre qué puede haber en un laberinto y si el juego tiene uno o varios jugadores. En vez de eso, nos centraremos en cómo se crean los laberintos. Definimos un laberinto como un conjunto de habitaciones. Una habitación conoce sus lindes; éstos pueden ser otra habitación, una pared o una puerta a otra habitación.

Las clases Habitación, Puerta y Pared definen los componentes del laberinto usado en todos nuestros ejemplos. Definiremos sólo las partes de esas clases que son importantes para crear un laberinto. Obviaremos los jugadores, las operaciones de visualización y aquéllas para caminar por el laberinto, además de otras funcionalidades importantes que no son relevantes para construir el laberinto.

El diagrama de la página siguiente muestra la relación entre estas clases.

Cada habitación tiene cuatro lados. Usaremos una enumeración Dirección en las implementaciones en C++ para especificar los lados norte, sur, este y oeste de una habitación:

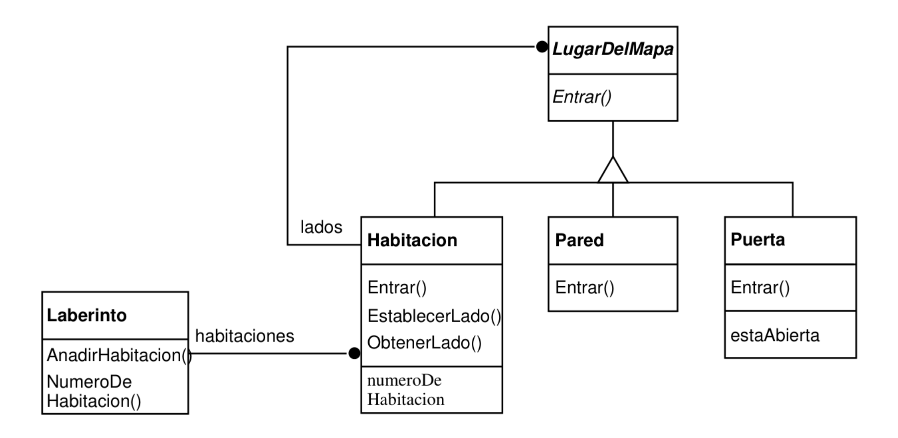
```
enum Direccion (Norte, Sur, Este, Oeste);
```

Las implementaciones en Smalltalk usan los símbolos correspondientes para representar estas direcciones.

La clase LugarDelMapa es la clase abstracta común de todos los componentes de un laberinto. Para simplificar el ejemplo, LugarDelMapa sólo define una operación, Entrar. Su significado

depende de en dónde estemos entrando. Si entramos en una habitación, cambiará nuestra posición. Si tratamos de entrar en una puerta, puede pasar una de estas dos cosas: si la puerta está abierta, pasaremos a la siguiente habitación; si está cerrada, nos daremos con ella en la nariz.

```
class LugarDelMapa {
public:
  virtual void Entrar() = 0;
};
```



Entrar proporciona una base para operaciones más sofisticadas del juego. Por ejemplo, si estamos en una habitación y decimos "Ir hacia el Este", el juego puede determinar fácilmente qué LugarDelMapa se encuentra inmediatamente al Este, para a continuación llamar a su operación Entrar. La operación Entrar de cada subclase concreta determinará si cambia nuestra posición o si nos golpeamos la nariz. En un juego real. Entrar podría tomar como argumento el objeto jugador que se está moviendo.

Habitación es la subclase concreta de LugarDelMapa que define las relaciones principales entre los componentes del laberinto. Mantiene referencias a otros objetos LugarDelMapa y guarda su número de habitación. Este número identificará las habitaciones en el laberinto.

```
class Habitacion : public LugarDelMapa {
public:
   Habitación(int numHabitacion);

   LugarDelMapa*        ObtenerLado(Direccion)
const;
   void             EstablecerLado(Dirección,
LugarDelMapa*);

   virtual void Entrar();

private:
   LugarDelMapa* _lados[4];
   int _numeroHabitacion;
}
```

Las clases siguientes representan la pared o la puerta que hay en cada lado de una estancia.

```
class Pared : public LugarDelMapa {
public:
   Pared();
   virtual void Entrar();
};

class Puerta : public LugarDelMapa {
   public:
      Puerta(Habitacion*  =  0,  Habitacion*  =
0);

      virtual void Entrar();
      Habitación* OtroLadoDe(Habitacion*);

   private:
      Room* _habitacion1;
      Room* _habitacion2;
      bool _estaAbierta;
   }
```

Necesitamos conocer más cosas además de las partes de un laberinto. Definiremos también una clase Laberinto que represente una serie de habitaciones. Laberinto también puede encontrar una determinada habitación usando su operación NumeroDeHabitacion.

```
class Laberinto {
public:
  Laberinto();

  void AnadirHabitacion(Habitacion*);
  Habitacion* NumeroDeHabitacion(int) const;
private:
  // …
};
```

NumeroDeHabitacion podría hacer una búsqueda usando una búsqueda lineal, una tabla de dispersión (*hash*) o incluso un simple array. Pero no nos preocuparemos aquí por esos detalles, sino que nos centraremos en cómo especificar los componentes de un objeto laberinto.

Otra clase que definimos es JuegoDelLaberinto, que es la que crea el laberinto. Una forma sencilla de crear el laberinto es con una serie de operaciones que añadan componentes a un laberinto y los conecten entre sí. Por ejemplo, las siguientes funciones miembro crearán un laberinto consistente en dos habitaciones con una puerta entre ambas:

```
Laberinto*  JuegoDelLaberinto::CrearLaberinto
() {
  Laberinto* unLaberinto = new Laberinto;
  Habitacion* h1 = new Habitacion(1);
  Habitacion* h2 = new Habitacion(2);
  Puerta* thePuerta = new Puerta(h1, h2);

  unLaberinto->AnadirHabitacion(h1);
  unLaberinto->AnadirHabitacion(h2);

  h1->EstablecerLado(Norte, new Pared);
  h1->EstablecerLado(Este, laPuerta);
  h1->EstablecerLado(Sur, new Pared);
  h1->EstablecerLado(Oeste, new Pared);
  h2->EstablecerLado(Norte, new Pared);
  h2->EstablecerLado(Este, new Pared);
  h2->EstablecerLado(Sur, new Pared);
  h2->EstablecerLado(Oeste, laPuerta);

  return unLaberinto;
}
```

Esta función es bastante complicada, teniendo en cuenta que todo lo que hace es crear un laberinto con dos habitaciones. Hay maneras obvias de hacerla más simple. Por ejemplo, el constructor de Habitacion podría inicializar por omisión los lados con paredes. Pero eso simplemente movería el código a otro lugar. El problema real de esta función miembro no es su tamaño, sino su *inflexibilidad*, al fijar en el código la distribución del laberinto. Cambiar la distribución significa cambiar esta función miembro, ya sea redefiniéndola —lo que significa reimplementarla en su totalidad— o cambiando partes de ella —lo que es propenso a errores y no promueve la reutilización—.

Los patrones de creación muestran cómo hacer este diseño más *flexible*, no necesariamente más pequeño. En concreto, harán que sea más fácil cambiar las clases que definen los componentes de un laberinto.

Supongamos que quisiéramos reutilizar la distribución de un laberinto existente para un nuevo juego que contiene laberintos encantados. El juego de los laberintos encantados tiene nuevos tipo de componentes, como PuertaQueNecesitaHechizo, un tipo de puerta que sólo puede cerrarse y abrirse con un hechizo; y HabitacionEncantada, una habitación que puede contener elementos no convencionales, tales como llaves mágicas o hechizos. ¿Cómo podemos cambiar CrearLaberinto fácilmente para crear laberintos con estas nuevas clases de objetos?

En este caso, el principal obstáculo para el cambio reside en fijar en el código las clases de las que se crean las instancias. Los patrones de creación proporcionan varias formas de eliminar las referencias explícitas a clases concretas en el código que necesita crear instancias de ellas:

- Si CrearLaberinto llama a funciones virtuales en vez de a constructores para crear las habitaciones, paredes y puertas que necesita, entonces podemos cambiar las clases de las instancias a crear haciendo una subclase de JuegoDelLaberinto y redefiniendo dichas funciones virtuales. Este enfoque es un ejemplo del patrón Factory Method (99).
- Si a CrearLaberinto le pasamos un objeto como parámetro a usar para crear las habitaciones, paredes y puertas, podemos

cambiar las clases de estos elementos pasándole un parámetro diferente. Éste es un ejemplo del patrón Abstract Factory (79).

- Si a CrearLaberinto le pasamos un objeto que puede crear un nuevo laberinto en su totalidad usando operaciones para añadir habitaciones, puertas y paredes al laberinto que construye, podemos usar la herencia para cambiar partes del laberinto o el modo en que éste es construido. Éste es un ejemplo del patrón Builder (89).
- Si parametrizamos CrearLaberinto con varias habitaciones, puertas y paredes prototípicas, las cuales copia y luego añade al laberinto, podemos cambiar la composición del laberinto sustituyendo estos objetos prototípicos por otros diferentes. Éste es un ejemplo del patrón Prototype 009).

El patrón de creación que falta, el Singleton (119), puede garantizar que sólo haya un laberinto por juego y que todos los objetos del juego puedan acceder a él —sin necesidad de acudir a variables o funciones globales—. El Singleton también hace más fácil ampliar o sustituir el laberinto sin tocar el código existente.

ABSTRACT FACTORY

(Fábrica Abstracta)

PROPÓSITO

Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.

TAMBIÉN CONOCIDO COMO

Kit

MOTIVACIÓN

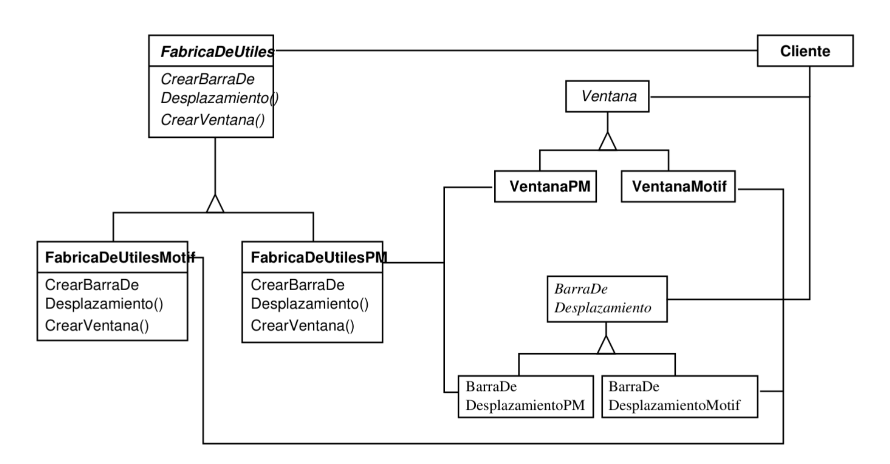
Pensemos en un toolkit de interfaces de usuario que admita múltiples estándares de interfaz de usuario [30], tales como Motif y Presentation Manager. Los distintos estándares interfaz de usuario definen distintos aspectos y formas de comportamiento de los "útiles" [31] de la interfaz de usuario, como las barras de desplazamiento, ventanas y botones. Para que una aplicación pueda portarse a varios estándares de interfaz de usuario, ésta no debería codificar sus útiles para una interfaz de usuario en particular. Si la aplicación crea instancias de clases o útiles específicos de la interfaz de usuario será difícil cambiar ésta más tarde.

Podemos solucionar este problema definiendo una clase abstracta FabricaDeUtiles que declara una interfaz para crear cada tipo básico de útil (*widget*). También hay una clase abstracta para cada tipo de útil, y las subclases concretas implementan útiles para un estándar concreto de interfaz de usuario. La interfaz de FabricaDeUtiles tiene una operación que devuelve un nuevo objeto para cada clase abstracta de útil. Los clientes llaman a estas operaciones para obtener instancias de útiles, pero no son

conscientes de las clases concretas que están usando. De esta manera los clientes son independientes de la interfaz de usuario.

Hay una subclase concreta de FabricaDeUtiles para cada estándar de interfaz de usuario. Cada subclase implementa las operaciones que crean el útil apropiado para su interfaz de usuario. Por ejemplo, la operación CrearBarraDeDesplazamiento de la FabricaDeUtilesMotif crea y devuelve una instancia de una barra de desplazamiento Motif, mientras que la misma operación en FabricaDeUtilesPM devuelve una barra de desplazamiento para Presentation Manager. Los clientes crean útiles únicamente a través de la interfaz FabricaDeUtiles y no tienen conocimiento de las clases que implementan los útiles para una determinada interfaz de usuario. En otras palabras, los clientes no tienen que atarse a una clase concreta, sino sólo a una interfaz definida por una clase abstracta.

Una FabricaDeUtiles también fuerza a que se cumplan las dependencias entre las clases concretas de útiles. Una barra de desplazamiento Motif debería usarse con un botón Motif y un editor de texto Motif, y esa restricción se cumple automáticamente como consecuencia de usar una FabricaDeUtilesMotif.



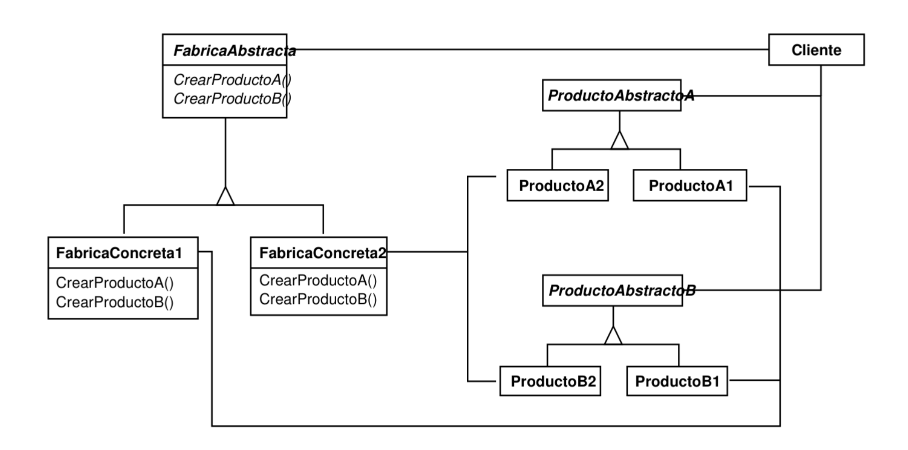
APLICABILIDAD

Úsese el patrón Abstract Factory cuando

- un sistema debe ser independiente de cómo se crean, componen y representan sus productos.

- un sistema debe ser configurado con una familia de productos de entre varias.
- una familia de objetos producto relacionados está diseñada para ser usada conjuntamente, y es necesario hacer cumplir esta restricción.
- quiere proporcionar una biblioteca de clases de productos, y sólo quiere revelar sus interfaces, no sus implementaciones.

ESTRUCTURA



PARTICIPANTES

- **FabricaAbstracta** (FabricaDeUtiles)
  - declara una interfaz para operaciones que crean objetos producto abstractos.
- **FabricaConcreta** (FabricaDeUtilesMotif, FabricaDeUtilesPM)
  - implementa las operaciones para crear objetos producto concretos.
- **Producto Abstracto** (Ventana, BarraDeDesplazamiento)
  - declara una interfaz para un tipo de objeto producto.
- **ProductoConcreto** (VentanaMotif, BarraDeDesplazamientoMotif)
  - define un objeto producto para que sea creado por la

fábrica correspondiente.
    - implementa la interfaz Producto Abstracto.
- **Cliente**
    - sólo usa interfaces declaradas por las clases FabricaAbstracta y ProductoAbstracto.

COLABORACIONES

- Normalmente sólo se crea una única instancia de una clase FabricaConcreta en tiempo de ejecución. Esta fábrica concreta crea objetos producto que tienen una determinada implementación. Para crear diferentes objetos producto, los clientes deben usar una fábrica concreta diferente.
- FabricaAbstracta delega la creación de objetos producto en su subclase FabricaConcreta.

CONSECUENCIAS

El patrón Abstract Factory tiene las siguientes ventajas e inconvenientes:

1. *Aísla las clases concretas*. El patrón Abstract Factory ayuda a controlar las clases de objetos que crea una aplicación. Como una fábrica encapsula la responsabilidad y el proceso de creación de objetos producto, aísla a los clientes de las clases de implementación. Los clientes manipulan las instancias a través de sus interfaces abstractas. Los nombres de las clases producto quedan aisladas en la implementación de la fábrica concreta; no aparecen en el código cliente.
2. *Facilita el intercambio de familias de productos*. La clase de una fábrica concreta sólo aparece una vez en una aplicación —cuando se crea—. Esto facilita cambiar la fábrica concreta que usa una aplicación. Como una fábrica abstracta crea una familia completa de productos, toda la familia de productos cambia de una vez. En nuestro ejemplo de la interfaz de usuario, podemos cambiar de útiles Motif a útiles Presentation Manager simplemente cambiando los correspondientes objetos fábrica y volviendo a crear la

interfaz.

3. *Promueve la consistencia entre productos*. Cuando se diseñan objetos producto en una familia para trabajar juntos, es importante que una aplicación use objetos de una sola familia a la vez. FabricaAbstracta facilita que se cumpla esta restricción.
4. *Es difícil dar cabida a nuevos tipos de productos*. Ampliar las fábricas abstractas para producir nuevos tipos de productos no es fácil. Esto se debe a que la interfaz FabricaAbstracta fija el conjunto de productos que se pueden crear. Permitir nuevos tipos de productos requiere ampliar la interfaz de la fábrica, lo que a su vez implica cambiar la clase FabricaAbstracta y todas sus subclases. En la sección de Implementación se analiza una solución a este problema.

IMPLEMENTACIÓN

Éstas son algunas técnicas útiles para implementar el patrón Abstract Factory.

1. *Fábricas únicas*. Normalmente una aplicación sólo necesita una instancia de una FabricaConcreta por cada familia de productos. Por tanto, suele implementarse mejor como un Singleton (119).

2. *Crear los productos*. FabricaAbstracta sólo declara una *interfaz* para crear productos. Se deja a las subclases ProductoConcreto el crearlos realmente. El modo más común de hacer esto es definiendo un método de fabricación para cada producto (*véase* el patrón Factory Method (99)). Una fábrica concreta especificará sus productos redefiniendo el método fábrica de cada uno. Si bien esta implementación es sencilla, requiere una nueva subclase fábrica concreta para cada familia de productos, incluso aunque las familias de productos difieran i sólo ligeramente.

   En caso de que sea posible tener muchas familias de productos, la fábrica concreta puede implementarse

usando el patrón Prototype (109). La fábrica concreta se inicializa con una instancia prototípica de cada producto de la familia, y crea un nuevo producto clonando su prototipo. El enfoque basado en prototipos elimina la necesidad de una nueva clase de fábrica concreta para cada nueva familia de productos.

A continuación presentamos un modo de implementar una fábrica basada en prototipos en Smalltalk. La fábrica concreta guarda los prototipos a clonar en un diccionario llamado cata logoDePartes. El método hacer: obtiene el prototipo y lo clona:

```
hacer: nombreParte
     ^     (catalogoDePartes     at:
nombreParte) copy
```

La fábrica concreta tiene un método para añadir partes al catálogo.

```
anadirParte:  plantillaParte  nombre:
nombreParte
     catalogoDePartes            at:
nombreParte put: plantillaParte
```

Los prototipos se añaden a la fábrica identificándolos con un símbolo:

```
unaFabrica anadirParte:
   unPrototipo nombre: #UtilACME
```

Es posible una variación del enfoque basado en prototipos en lenguajes que tratan a las clases como objetos en toda regla (Smalltalk y Objective C, por ejemplo). En tales lenguajes podemos pensar en una clase como una fábrica degenerada que sólo crea un tipo de producto. Podemos almacenar *clases* en variables dentro de una fábrica concreta que crea los distintos productos concretos, de

manera muy parecida a los prototipos. Estas clases crean nuevas instancias en nombre de la fábrica concreta. Definimos una nueva fábrica inicializando una instancia de una fábrica concreta con *clases* de productos en vez de mediante subclases. Este enfoque se aprovecha de características del lenguaje, mientras que el enfoque basado en prototipos puro es independiente del lenguaje.

Al igual que la fábrica basada en prototipos en Smalltalk que acabamos de ver, la versión basada en clases tendrá una única variable de instancia catalogoDePartes, que es un diccionario cuya clave es el nombre de la parte. En vez de guardar los prototipos a ser clonados, catalogoDePartes almacena las clases de los productos. El método hacer: quedaría ahora así:

```
hacer: nombreParte
     ^     (catalogoDePartes     at:
nombreParte) new
```

3. *Definir fabricas extensibles*. Fabrica Abstracta por lo general define una operación diferente para cada tipo de producto que puede producir. Los tipos de producto están codificados en las signaturas de las operaciones. Añadir un nuevo tipo de producto requiere cambiar la interfaz de FabricaAbstracta y todas las clases que dependen de ella.

   Un diseño más flexible, aunque menos seguro, es añadir un parámetro a las operaciones que crean objetos. Este parámetro especifica el tipo de objeto a ser creado. Podría tratarse de un identificador de clase, un entero, una cadena de texto o cualquier otra cosa que identifique el tipo de producto. De hecho, con este enfoque, FabricaAbstracta sólo necesita una única operación "Hacer" con un parámetro que indique el tipo de objeto a crear. Ésta es la técnica usada en las fábricas abstractas basadas en clases y en prototipos que se examinaron anteriormente. Esta variación es más fácil de usar en un

lenguaje dinámicamente tipado, como Smalltalk, que en uno estáticamente tipado, como C++. Podemos aplicarla en C++ sólo cuando todos los objetos tienen la misma clase base abstracta o cuando los objetos producto pueden ser convertidos con seguridad al tipo correcto por el objeto que los solicita. La sección de implementación del patrón Factory Method (99) muestra cómo implementar dichas operaciones parametrizadas en C++. Pero incluso cuando no es necesaria la conversión de tipos, todavía subyace un problema inherente: todos los productos se devuelven al cliente con la *misma* interfaz abstracta que el tipo de retorno. El cliente no podrá por tanto distinguir o hacer suposiciones seguras acerca de la clase de un producto. En caso de que los clientes necesiten realizar operaciones específicas de las subclases, éstas no estarán accesibles a través de la interfaz abstracta. Aunque el cliente podría hacer una conversión al tipo de una clase hija (esto es, un *downcast*) (por ejemplo, con dynamic_cast en C++), eso no siempre resulta viable o seguro, porque la conversión de tipos puede fallar. Éste es el inconveniente típico de una interfaz altamente flexible y extensible.

CÓDIGO DE EJEMPLO

Aplicaremos el patrón Abstract Factory para crear los laberintos de los que hablamos al principio de este capítulo.

La clase FabricaDeLaberintos puede crear los componentes de los laberintos. Construye habitaciones, paredes y puertas entre las habitaciones. Podría ser usada por un programa que lee de un fichero los planos de los laberintos y construye el correspondiente laberinto. O tal vez sea usada por un programa que construye los laberintos al azar. Los programas que construyen laberintos tornan una FabricaDeLaberintos como argumento, de manera que el programador puede especificar las clases de habitaciones, paredes y puertas a construir.

```
class FabricaDeLaberintos {
public:
   FabricaDeLaberintos();
```

```
      virtual   Laberinto*   HacerLaberinto()
const
         { return new Laberinto; }
      virtual Pared* HacerPared() const
         { return new Pared; }
      virtual  Habitacion*  HacerHabitacion(int
n) const
         { return new Habitacion(n); }
      virtual  Puerta*  HacerPuerta(Habitacion*
h1,
                     Habitacion* h2) const
         { return new Puerta(h1, h2); }
};
```

Recordemos que la función CrearLaberinto construye un pequeño laberinto consistente en dos habitaciones con una puerta entre ellas. CrearLaberinto fija en el código los nombres de clases, dificultando así la posibilidad de crear laberintos con otros componentes.

Presentamos una versión de CrearLaberinto que remedia esta deficiencia tomando como parámetro una FabricaDeLaberintos:

```
Laberinto*  JuegoDelLaberinto::CrearLaberinto
(
                    FabricaDeLaberintos&
fabrica) {
        Laberinto*       unLaberinto        *
fabrica.HacerLaberinto();
        Habitacion*           h1            =
fabrica.HacerHabitacion(1);
        Habitacion*           h2            =
fabrica.HacerHabitacion(2);
        Puerta*          unaPuerta          =
fabrica.HacerPuerta(h1, h2);

        unLaberinto->AnadirHabitacion(h1);
        unLaberinto->AnadirHabitacion(h2);

        h1->EstablecerLado(Norte,
fabrica.HacerPared());
        h1->EstablecerLado(Este, unaPuerta);
        h1->EstablecerLado(Sur,
fabrica.HacerPared());
        h1->EstablecerLado(Oeste,
fabrica.HacerPared());
```

```
        h2->EstablecerLado(Norte,
fabrica.HacerPared());
        h2->EstablecerLado(Este,
fabrica.HacerPared());
        h2->EstablecerLado(Sur,
fabrica.HacerPared());
        h2->EstablecerLado(Oeste, unaPuerta);

        return unLaberinto;
   }
```

Podemos crear FabricaDeLaberintosEncantados, una fábrica para laberintos encantados, como una subclase de FabricaDeLaberintos. FabricaDeLaberintosEncantados redefinirá diferentes funciones miembro y devolverá diferentes subclases de Habitación, Pared, etc.

```
class FabricaDeLaberintosEncantados : public
FabricaDeLaberintos {
public:
   FabricaDeLaberintosEncantados();

   virtual Habitacion* HacerHabitacion(int n)
const
      { return new HabitacionEncantada(n,
Hechizar()); }

   virtual Puerta* HacerPuerta(Habitacion*
h1,
                    Habitacion* h2) const
      { return new PuertaQueNecesitaHechizo
(h1, h2); }

protected:
   Hechizo* Hechizar() const;
};
```

Supongamos ahora que queremos hacer un juego del laberinto en el que una habitación puede tener puesta una bomba. Si la bomba explota, como mínimo dañará las paredes. Podemos hacer una subclase de Habitación que compruebe si la habitación tiene una bomba y si ha explotado. También necesitamos una subclase de Pared para saber el daño causado en ella. Llamaremos a estas clases HabitacionConUnaBomba y ParedExplosionada.

La última clase que definiremos es FabricaDeLaberintosConBombas, una subclase de FabricaDeLaberintos que garantiza que las paredes son de la clase ParedExplosionada y que las habitaciones son de la clase HabitacionConBomba. FabricaDeLaberintosConBombas sólo necesita redefinir dos funciones:

```
Pared*
FabricaDeLaberintosConBombas::HacerPared   ()
const {
          return new ParedExplosionada;
     }

     Habitacion*
FabricaDeLaberintosConBombas::HacerHabitacion(int
n)
                    const {
          return new HabitacionConBomba(n);
     }
```

Para construir un laberinto que pueda contener bombas, simplemente llamamos a CrearLaberinto con una FabricaDeLaberintosConBombas.

```
JuegoDelLaberinto juego;
FabricaDeLaberintosConBombas fabrica;

juego.CrearLaberinto(fabrica);
```

CrearLaberinto puede recibir igualmente una instancia de FabricaDeLaberintosEncantados para construir laberintos encantados.

Nótese que FabricaDeLaberintos no es más que una colección de métodos de fabricación. Ésta es la forma más normal de implementar el patrón Abstract Factory. Además, FabricaDeLaberintos no es una clase abstracta, de manera que hace tanto de FabricaAbstracta *como* de FabricaConcreta. También ésta es la implementación más común del patrón Abstract Factory para aplicaciones sencillas. Al ser FabricaDeLaberintos una clase

concreta que consiste solamente en métodos de fabricación, es fácil hacer una nueva FabricaDeLaberintos creando una subclase y redefiniendo las operaciones que se necesite cambiar. CrearLaberinto usa la operación EstablecerLado de las habitaciones para especificar sus lados. Si se crean habitaciones con un FabricaDeLaberintosConBombas el laberinto estará formado por objetos HabitacionConBomba que tendrán objetos ParedExplosionada como lados. En caso de que HabitacionConBomba tuviera que acceder a un miembro de ParedExplosionada especifico de la subclase, tendría que convertir una referencia a sus paredes de Pared* a ParedExplosionada*. Esta conversión de tipos (*downcasting*) es seguro siempre y cuando el argumento *sea* realmente una ParedExplosionada, lo que está garantizado si las paredes se construyen únicamente mediante una FabricaDeLaberintosConBombas.

Por supuesto, en el caso de los lenguajes dinámicamente tipados, como Smalltalk, no es necesaria la conversión de tipos, pero podrían producirse errores en tiempo de ejecución si encuentran una Pared donde esperaban una *subclase* de Pared. Usar una Fábrica Abstracta para crear las paredes ayuda a evitar estos errores de tiempo de ejecución al garantizar que sólo se pueden crear ciertos tipos de paredes.

Veamos una versión de FabricaDeLaberintos en Smalltalk, con una única operación hacer que recibe como parámetro el tipo de objeto a construir. Además, la fábrica concreta almacena las clases de los productos que crea.

En primer lugar, escribiremos un CrearLaberinto equivalente en Smalltalk:

```
CrearLaberinto: unaFabrica
     ¦ habitacion1 habitacion2 unaPuerta ¦
          Habitacion1  :=  (unaFabrica  hacer:
#habitacion) numero: 1.
          habitacion2  :=  (unaFabrica  hacer:
#habitacion) numero: 2.
          unaPuerta   :=   (unaFabrica   hacer:
#puerta) de:
          habitacion1 a: habitacion2.
          Habitacion1   lado:   #norte   put:
(unaFabrica hacer: #pared).
```

```
          Habitacion1   lado:   #este   put:
unaPuerta.
          Habitacion1   lado:   #sur   put:
(unaFabrica hacer: #pared).
          Habitacion1   lado:   #oeste   put:
(unaFabrica hacer: #pared).
          habitacion2   lado:   #norte   put:
(unaFabrica hacer: #pared).
          habitacion2   lado:   #este   put:
(unaFabrica hacer: #pared).
          habitacion2   lado:   #sur   put:
(unaFabrica hacer: #pared).
          habitacion2   lado:   #oeste   put:
unaPuerta.
          ^  Laberinto  new  anadirHabitacion:
habitacion1;
anadirHabitacion: habitacion2; yourself
```

Como ya dijimos en la sección de Implementación, FabricaDeLaberintos sólo necesita una única variable de instancia catalogoDePartes, que consiste en un diccionario cuya clave es la clase del componente. Recordemos también cómo implementamos el método hacer::

```
hacer: nombreParte
     ^ (catalogoDePartes at: nombreParte) new
```

Ahora podemos crear una FabricaDeLaberintos y usarla para implementar crearLaberinto. Crearemos la fábrica usando el método crearFabricaDeLaberintos de la clase JuegoDelLaberinto.

```
crearFabricaDeLaberintos
     ^ (FabricaDeLaberintos new
        anadirParte: Pared nombre: #pared;
        anadirParte:    Habitacion    nombre:
#habitacion;
        anadirParte: Puerta nombre: #puerta;
        yourself)
```

Una FabricaDeLaberintosConBombas o una

FabricaDeLaberintosEncantados se crean asociando diferentes clases con las claves. Por ejemplo, un FabricaDeLaberintosEncantados se podría crear como sigue:

```
crearFabricaDeLaberintos
     ^ (FabricaDeLaberintos new
        anadirParte: Pared nombre: #pared;
        anadirParte:      HabitacionEncantada
nombre: #habitacion;
        anadirParte:  PuertaQueNecesitaHechizo
nombre: #puerta;
        yourself)
```

USOS CONOCIDOS

Interviews usa el sufijo "Kit" [Lin92] para denotar las clases FabricaAbstracta. Define las fábricas abstractas WidgetKit y DialogKit para generar objetos específicos de la interfaz de usuario. Interviews también incluye un LayoutKit que genera diferentes objetos de composición dependiendo de la disposición deseada. Así por ejemplo, una distribución horizontal puede necesitar diferentes objetos de composición dependiendo de la orientación del documento (vertical o apaisado).

ET++ [WGM88J usa el patrón Abstract Factory para lograr portabilidad entre varios sistemas de ventanas (por ejemplo, X Windows y Sun View). La clase base abstracta WindowSystem define la interfaz para crear objetos que representan recursos del sistema de ventanas (por ejemplo, MakeWindow, MakeFont, MakeColor). Las subclases concretas implementan las interfaces para un determinado sistema de ventanas. En tiempo de ejecución, ET++ crea una instancia de una subclase concreta de WindowSystem que es la encargada de crear objetos concretos de recursos del sistema.

PATRONES RELACIONADOS

Las clases FabricaAbstracta suelen implementarse con métodos de fabricación (patrón Factory Method (99)), pero también se pueden implementar usando prototipos (patrón Prototype (109)).

Una fábrica concreta suele ser un Singleton (119).

BUILDER (Constructor)

PROPÓSITO

Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.

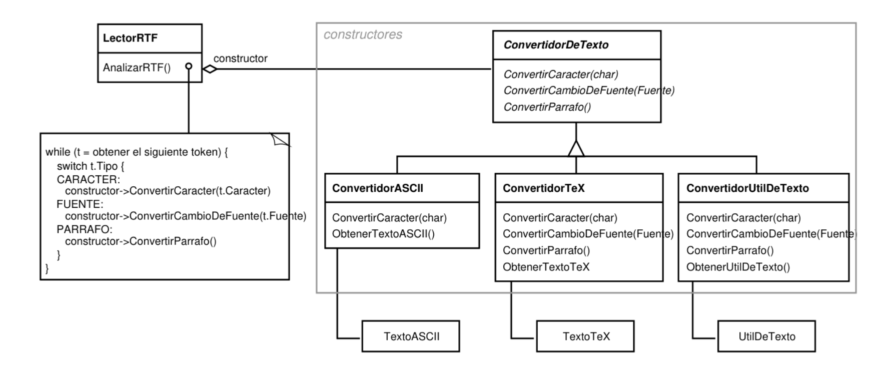
MOTIVACIÓN

Un lector del formato de intercambio de documentos RTF (*Rich Text Format*) debería poder convertir RTF a muchos formatos de texto. Podría convertir documentos RTF a texto ASCII o a un útil [32] de texto que pueda editarse de forma interactiva. El problema, no obstante, es que el número de conversiones posibles es indefinido. Por tanto, tendría que ser fácil añadir una nueva conversión sin modificar el lector.

Una solución es configurar la clase LectorRTF con un objeto ConvertidorDeTexto que convierta RTF a otra representación textual. Cuando el LectorRTF analiza el documento RTF, usa el ConvertidorDeTexto para realizar la conversión. Cada vez que el LectorRTF reconozca un token RTF (ya sea texto normal o una palabra de control de RTF), envía una petición al ConvertidorDeTexto para que lo convierta. Los objetos ConvertidorDeTexto son responsables de realizar la conversión de datos y de representar el token en un determinado formato.

Las subclases de ConvertidorDeTexto están especializadas en diferentes conversiones y formatos. Por ejemplo, un ConvertidorASCII hace caso omiso de las peticiones de conversión de cualquier otra cosa que no sea texto sin formato. Por otro lado, un ConvertidorTeX, implementará operaciones para todas las peticiones, con el objetivo de producir una representación en TeX con toda la información de estilo que haya en el texto. Un

ConvertidorUtilDeTexto producirá un objeto complejo de interfaz de usuario que permita al usuario ver y editar el texto.



La clase de cada tipo de convertidor toma el mecanismo de creación y ensamblaje de un objeto complejo y lo oculta tras una interfaz abstracta. El convertidor se separa del lector, que es el responsable de analizar un documento RTF.

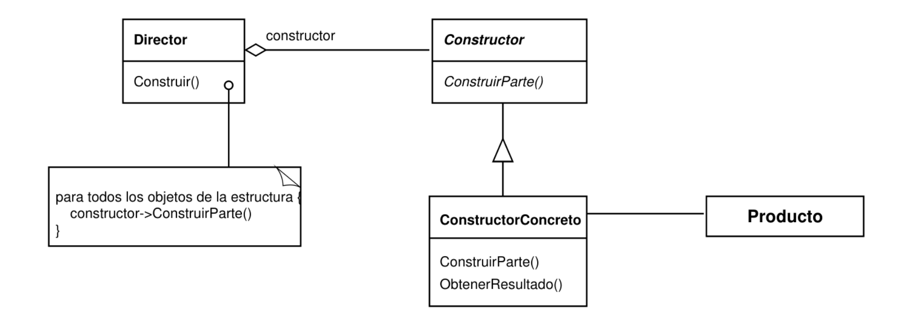
El patrón Builder expresa todas estas relaciones. Cada clase de convertidor se denomina **constructor**, en el contexto de este patrón, y al lector se le llama **director**. Aplicado a este ejemplo, el patrón Builder separa el algoritmo para interpretar un formato textual (es decir, el analizador de documentos RTF) de la manera en que se crea y se representa el formato de destino. Esto permite reutilizar el algoritmo de análisis de LectorRTF para crear diferentes representaciones de texto a partir de documentos RTF —basta con configurar el LectorRTF con diferentes subclases de ConvertidorDeTexto—.

APLICABILIDAD

Úsese el patrón Builder cuando

- el algoritmo para crear un objeto complejo debiera ser independiente de las partes de que se compone dicho objeto y de cómo se ensamblan.
- el proceso de construcción debe permitir diferentes representaciones del objeto que está siendo construido.

ESTRUCTURA



PARTICIPANTES

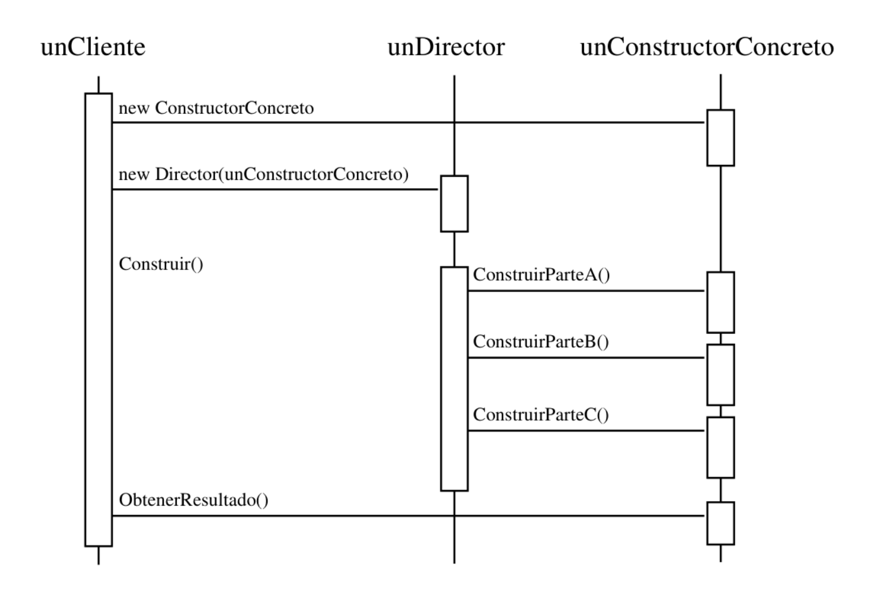
- **Constructor** (ConvertidorDeTexto)
  - especifica una interfaz abstracta para crear las partes de un objeto Producto.
- **Constructor Concreto** (ConvertidorASCII, ConvertidorTeX, ConvertidorUtilDeTexto)
  - implementa la interfaz Constructor para construir y ensamblar las partes del producto.
  - define la representación a crear.
  - proporciona una interfaz para devolver el producto (p. ej., ObtenerTextoASCII, ObtenerUtilDeTexto).
- **Director** (LectorRTF)
  - construye un objeto usando la interfaz Constructor.
- **Producto** (TextoASCII, TextoTeX, UtilDeTexto)
  - representa el objeto complejo en construcción. El ConstructorConcreto construye la representación interna del producto y define el proceso de ensamblaje.
  - incluye las clases que definen sus partes constituyentes, incluyendo interfaces para ensamblar las partes en el resultado final.

COLABORACIONES

- El cliente crea el objeto Director y lo configura con el objeto Constructor deseado.

- El Director notifica al constructor cada vez que hay que construir una parte de un producto.
- El Constructor maneja las peticiones del director y las añade al producto.
- El cliente obtiene el producto del constructor.

El siguiente diagrama de interacción ilustra cómo cooperan con un cliente el Constructor y el Director.



CONSECUENCIAS

Éstas son las principales consecuencias del patrón Builder:

1. *Permite variar la representación interna de un producto*. El objeto Constructor proporciona al director una interfaz abstracta para construir el producto. La interfaz permite que el constructor oculte la representación y la estructura interna del producto. También oculta el modo en que éste es ensamblado. Dado que el producto se construye a través de una interfaz abstracta, todo lo que hay que hacer para cambiar la representación interna del producto es definir un nuevo tipo de constructor.
2. *Aísla el código de construcción y representación*. El patrón

Builder aumenta la modularidad al encapsular cómo se construyen y se representan los objetos complejos. Los clientes no necesitan saber nada de las clases que definen la estructura interna del producto; dichas clases no aparecen en la interfaz del Constructor. Cada ConstructorConcreto contiene todo el código para crear y ensamblar un determinado tipo de producto. El código sólo se escribe una vez; después, los diferentes Directores pueden reutilizarlo para construir variantes de Producto a partir del mismo conjunto de partes. En el ejemplo anterior de RTF, podríamos definir un lector para otro formato distinto de RTF. como por ejemplo un LectorSGML, y usar los mismos objetos ConvertidorDeTexto para generar representaciones TextoASCII, TextoTeX y UtilDeTexto de documentos SGML.

3. *Proporciona un control más fino sobre el proceso de construcción.* A diferencia de los patrones de creación que construyen los productos de una vez, el patrón Builder construye el producto paso a paso, bajo el control del director. El director sólo obtiene el producto del constructor una vez que éste está terminado. Por tanto, la interfaz Constructor refleja el proceso de construcción del producto más que otros patrones de creación. Esto da un control más fino sobre el proceso de construcción y, por tanto, sobre la estructura interna del producto resultante.

IMPLEMENTACIÓN

Normalmente hay una clase abstracta Builder que define una operación para cada componente que puede ser creado. La implementación predeterminada de estas operaciones no hace nada. Una clase ConstructorConcreto redefine las operaciones para los componentes que está interesado en crear.

Éstas son otras cuestiones de implementación que hay que considerar:

1. *Interfaz de ensamblaje y construcción.* Los constructores construyen sus productos paso a paso. Por tanto, la interfaz de la clase Constructor debe ser lo suficientemente

general como para permitir construir productos por parte de todos los tipos de constructores concretos.

Una cuestión de diseño fundamental tiene que ver con el modelo del proceso de construcción y ensamblaje. Normalmente basta con un modelo según el cual los resultados de las peticiones de construcción simplemente se van añadiendo al producto. En el ejemplo del RTF, el constructor convierte y añade el siguiente token al texto que ha convertido hasta la fecha.

Pero a veces podríamos necesitar acceder a las partes del producto que ya fueron construidas. En el ejemplo del laberinto que presentamos en el Código de Ejemplo, la interfaz ConstructorLaberinto permite añadir una puerta entre habitaciones existentes. Otro ejemplo son las estructuras arbóreas, como los árboles sintácticos que se crean de abajo a arriba. En ese caso, el constructor devolvería nodos hijos al director, el cual los devolvería al constructor para construir los nodos padre.

2. ¿*Por qué no usar clases abstractas para los productos*? En general, los productos creados por los constructores concretos tienen representaciones tan diferentes que sería de poca ayuda definir una clase padre común para los diferentes productos. En el ejemplo del RTF, es poco probable que los objetos TextoASCII y UtilDeTexto tengan una interfaz común. Como el cliente suele configurar al director con el constructor concreto adecuado, sabe qué subclase concreta de Constructor se está usando, y puede manejar sus productos en consecuencia.
3. *Métodos vacíos de manera predeterminada en el constructor*. En C++, los métodos de creación no se declaran como funciones miembro virtuales puras a propósito. En vez de eso, se definen como métodos vacíos, lo que permite que los clientes redefinan sólo las operaciones en las que están interesados.

CÓDIGO DE EJEMPLO

Definiremos una variante de la función miembro CrearLaberinto (páginas
76-77)
que toma como argumento un constructor de la clase ConstructorLaberinto.

La clase ConstructorLaberinto define la siguiente interfaz para construir laberintos:

```
class ConstructorLaberinto {
public:
  virtual void ConstruirLaberinto() { }
  virtual    void    ConstruirHabitacion(int
habitacion) { }
  virtual      void      ConstruirPuerta(int
habitacionDesde, int habitacionHasta) { }

  virtual Maze* ObtenerLaberinto() { return
0; }
protected:
  ConstructorLaberinto();
};
```

Esta interfaz puede crear tres cosas: (1) el laberinto, (2) habitaciones con un determinado número de habitación y (3) puertas entre habitaciones numeradas. La operación ObtenerLaberinto devuelve el laberinto al cliente. Las subclases de ConstructorLaberinto redefinirán esta operación para devolver el laberinto que construyen.

Todas las operaciones para construir el laberinto de ConstructorLaberinto por omisión no hacen nada. No se declaran como virtuales puras para permitir que las clases derivadas redefinan sólo aquellos métodos en los que estén interesadas.

Dada la interfaz de ConstructorLaberinto, podemos cambiar la función miembro CrearLaberinto para que tome como parámetro este constructor.

```
Laberinto*  JuegoDelLaberinto::CrearLaberinto
(ConstructorLaberinto& constructor)
{
constructor.ConstruirLaberinto();
```

```
constructor.ConstruirHabitacion(1);
constructor.ConstruirHabitacion(2);
constructor.ConstruirPuerta(1, 2);

return constructor.ObtenerLaberinto();
}
```

Compare esta versión de CrearLaberinto con la original. Fíjese en cómo el constructor oculta la representación interna de Laberinto —esto es, las clases que definen habitaciones, puertas y paredes— y cómo se ensamblan estas partes para completar el laberinto final. Alguien podría preguntarse por qué hay clases que representan habitaciones y puertas, pero no hay rastro de ninguna para las paredes. Esto facilita la manera de representar un laberinto, ya que no hay que cambiar ninguno de los clientes de ConstructorLaberinto.

Al igual que con los otros patrones de creación, el patrón Builder encapsula cómo se crean los objetos, en este caso a través de la interfaz definida por ConstructorLaberinto. Eso significa que podemos reutilizar ConstructorLaberinto para construir diferentes tipos de laberintos. Un ejemplo de esto es la operación CrearLaberintoComplejo:

```
Laberinto*
JuegoDelLaberinto::CrearLaberintoComplejo
(ConstructorLaberinto& constructor) {
constructor.ConstruirHabitacion(1);
// …
constructor.ConstruirHabitacion(1001);

return constructor.ObtenerLaberinto();
}
```

Nótese que ConstructorLaberinto no crea el laberinto en sí; su principal propósito es simplemente definir una interfaz para crear laberintos. Define implementaciones vacías más que nada por comodidad. Son las subclases de ConstructorLaberinto las que hacen el trabajo real.

La subclase ConstructorLaberintoEstandar es una

implementación que construye laberintos simples. Sabe qué laberinto está siendo creado gracias a la variable _laberintoActual.

```
class ConstructorLaberintoEstandar : public
ConstructorLaberinto {
public:
  ConstructorLaberintoEstandar));

  virtual void ConstruirLaberinto();
  virtual void ConstruirHabitacion(int);
  virtual void ConstruirPuerta(int, int);

  virtual Laberinto* ObtenerLaberinto();
private:
  Direccion        ParedNormal(Habitacion*,
Habitacion*);
  Laberinto* _laberintoActual;
};
```

ParedNormal es una operación auxiliar que calcula la dirección de la pared normal entre dos habitaciones.

El constructor de ConstructorLaberintoEstandar simplemente inicializa _laberintoActual.

```
ConstructorLaberintoEstandar::ConstructorLaberintoEstandar
() {
  _laberintoActual = 0;
}
```

ConstruirLaberinto crea una instancia de un Laberinto que otras operaciones ensamblarán y al final devolverán al cliente (mediante ObtenerLaberinto).

```
void
ConstructorLaberintoEstandar::ConstruirLaberinto
() {
  _laberintoActual = new Laberinto;
}

Laberinto*
```

```
ConstructorLaberintoEstandar::ObtenerLaberinto
() {
  return _laberintoActual;
}
```

La operación ConstruirHabitacion crea una habitación y construye sus paredes adyacentes:

```
void
ConstructorLaberintoEstandar::ConstruirHabitacion
(int n) {
  if                      (!_laberintoActual-
>NumeroDeHabitacion(n)) {
    Habitacion*    habitacion     =     new
Habitacion(n);
    _laberintoActual-
>AnadirHabitacion(habitacion);

    habitacion->EstablecerLado(Norte,    new
Pared);
    habitacion->EstablecerLado(Sur,      new
Pared);
    habitacion->EstablecerLado(Este,     new
Pared);
    habitacion->EstablecerLado(Oeste,    new
Pared);
  }
}
```

Para construir una puerta entre dos habitaciones, ConstructorLaberintoEstandar busca ambas habitaciones en el laberinto y encuentra su pared adyacente:

```
void
ConstructorLaberintoEstandar::ConstruirPuerta
(int n1, int n2) {
  Habitacion*    h1    =    _laberintoActual-
>NumeroDeHabitacion(n1);
  Habitacion*    h2    =    _laberintoActual-
>NumeroDeHabitacion(n2);
  Puerta* p = new Puerta(h1, h2);
```

```
    H1->EstablecerLado(ParedNormal(h1,h2), p);
    h2->EstablecerLado(ParedNormal(h2,h1), p);
}
```

Ahora los clientes pueden usar CrearLaberinto junto con ConstructorLaberintoEstandar para crear un laberinto:

```
Laberinto* laberinto;
JuegoDelLaberinto juego;
ConstructorLaberintoEstandar constructor;

juego.CrearLaberinto(constructor);
laberinto = constructor.ObtenerLaberinto();
```

Podríamos haber puesto todas las operaciones de ConstructorLaberintoEstandar en Laberinto y dejar que cada Laberinto se construyese a sí mismo. Pero al hacer a Laberinto más pequeña es más fácil entenderla y modificarla, y ConstructorLaberintoEstandar es fácil de separar de Laberinto. Lo que es más importante, separar ambas clases permite tener varios objetos ConstructorLaberinto, usando cada uno clases diferentes para las habitaciones, paredes y puertas.

Un ConstructorLaberinto más exótico es ContructorLaberintoContador. Este constructor no crea un laberinto; simplemente cuenta los distintos tipos de componentes que han sido creados.

```
class ContructorLaberintoContador : public
ConstructorLaberinto {
public:
    ContructorLaberintoContador();

    virtual void ConstruirLaberinto();
    virtual void ConstruirHabitacion(int);
    virtual void ConstruirPuerta(int, int);
    virtual void AnadirPared(int, Direccion);

    void ObtenerConteo(int&, int&) const;
private:
    int _puertas;
```

```
    int _habitaciones;
};
```

El constructor inicializa los contadores, y las operaciones redefinidas de ConstructorLaberinto los incrementan en consecuencia.

```
ContructorLaberintoContador::ContructorLaberintoContador
() {
    _habitaciones = _puertas = 0;
}

void
ContructorLaberintoContador::ConstruirHabitacion
(int) {
    _habitaciones++;
}

void
ContructorLaberintoContador::ConstruirPuerta
(int, int) {
    _puertas++;
}

void ContructorLaberintoContador::GetCounts (
    int& habitaciones, int& puertas
) const {
    habitaciones = _habitaciones;
    puertas = _puertas;
}
```

Así es como un cliente podría usar un ContructorLaberintoContador:

```
int habitaciones, puertas;
JuegoDelLaberinto juego;
ContructorLaberintoContador constructor;

juego.CrearLaberinto(constructor);
constructor.ObtenerConteo(habitaciones,
puertas);

cout « "El laberinto tiene "
        « habitaciones « " habitaciones y "
```

```
« puertas « " puertas" « endl;
```

USOS CONOCIDOS

La aplicación que convierte RTF es de ET++ [WGM88]. Su bloque de construcción de texto usa un constructor para procesar texto almacenado en formato RTF.

Builder es un patrón común en Smalltalk-80 [Par90]:

- La clase Parser en el subsistema del compilador es un Director que toma un objeto ProgramNodeBuildercomo argumento. Un objeto Parser notifica a su objeto ProgramNodeBuilder cada vez que reconoce una construcción sintáctica. Cuando finaliza el análisis, le pide al constructor el árbol de análisis sintáctico y se lo devuelve al cliente.
- ClassBuilder es un constructor que usan los objetos Class para crear subclases de sí mismas. En este caso un Class es tanto el Director como el Producto.
- ByteCodeStream es un constructor que crea un método compilado como un array de bytes. ByteCodeStream es un uso no estándar del patrón Builder, ya que el objeto complejo que construye está codificado como un array de bytes en vez de como un objeto Smalltalk normal. Pero la interfaz de ByteCodeStream es típica de un constructor, y sería fácil sustituir ByteCodeStream por una clase diferente que represente programas como un objeto compuesto.

El framework Service Configurator de Adaptive Communications Environment usa un constructor para construir componentes de servicio de red que están enlazados a un servidor en tiempo de ejecución [SS94], Los componentes se describen con un lenguaje de configuración que es analizado por un analizador LALR(l). Las acciones semánticas del analizador realizan operaciones sobre el constructor que añaden información al componente de servicios. En este caso, el analizador es el Director.

PATRONES RELACIONADOS

El patrón Abstract Factory (79) se parece a un Builder en que

también puede construir objetos complejos. La principal diferencia es que el patrón Builder se centra en construir un objeto complejo paso a paso. El Abstract Factory hace hincapié en familias de objetos producto (simples o complejos). El Builder devuelve el producto como paso final, mientras que el Abstract Factory lo devuelve inmediatamente.

Muchas veces lo que construye el constructor es un Composite (151).

FACTORY METHOD

(Método de Fabricación)

PROPÓSITO

Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.

TAMBIÉN CONOCIDO COMO

Virtual Constructor (Constructor Virtual)

MOTIVACIÓN

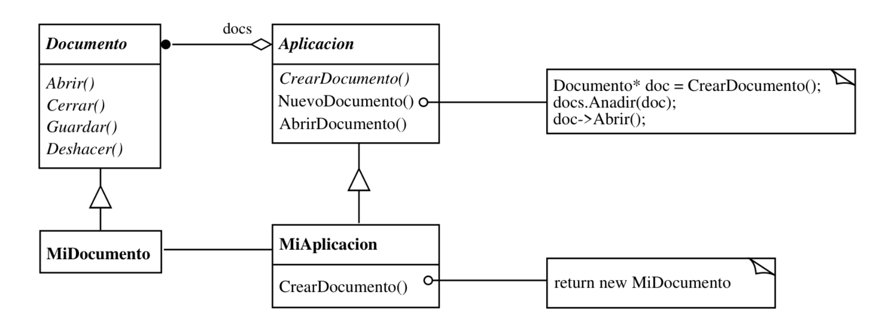
Los *frameworks* usan clases abstractas para definir y mantener relaciones entre objetos y también son muchas veces responsables de crear esos mismos objetos.

Pensemos en un framework de aplicaciones que pueda presentar varios documentos al usuario. Dos abstracciones clave de este framework son las clases Aplicación y Documento. Ambas son abstractas, y los clientes tienen que heredar de ellas para llevar a cabo sus implementaciones específicas de la aplicación. Por ejemplo, para crear una aplicación de dibujo, definimos las clases AplicacionDeDibujo y DocumentoDeDibujo. La clase Aplicación se encarga de gestionar Documentos y los creará cuando sea necesario, por ejemplo, cuando el usuario selecciona Abrir o Nuevo en un menú.

Dado que la subclase Documento concreta a instanciar es específica de la aplicación, la clase Aplicación no puede predecir qué subclase de Documento debe instanciar —la clase Aplicación

sólo sabe *cuándo* debería crearse un nuevo documento, no *qué tipo* de Documento crear—. Esto causa un dilema: el framework debe crear instancias de clases, pero sólo conoce clases abstractas, las cuales no pueden ser instanciadas.

El patrón Factory Method ofrece una solución al dilema. Encapsula el conocimiento acerca de qué subclases de Documento crear y saca ese conocimiento fuera del framework.



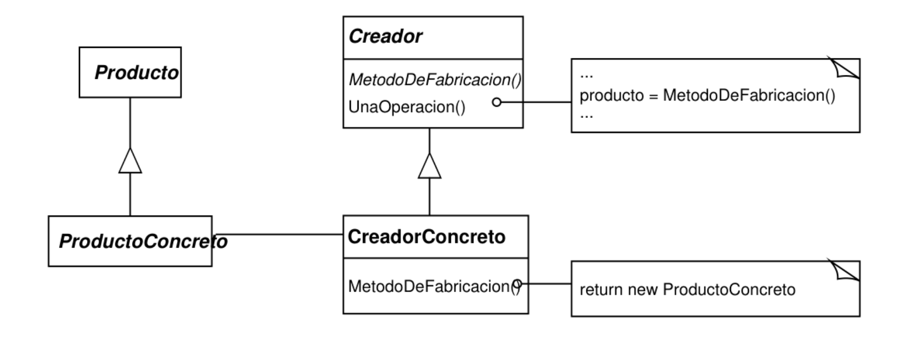
Las subclases de Aplicación redefinen su operación abstracta CrearDocumento para que devuelva la subclase de Documento adecuada. Una vez que se crea una instancia de la subclase de Aplicación, ésta puede crear instancias de Documentos específicos de la aplicación sin conocer sus clases. Llamaremos a CrearDocumento un método de fabricación porque es el responsable de “fabricar” un objeto.

APLICABILIDAD

Úsese el patrón Factory Method cuando

- una clase no puede prever la clase de objetos que debe crear.
- una clase quiere que sean sus subclases quienes especifiquen los objetos que ésta crea.
- las clases delegan la responsabilidad en una de entre varias clases auxiliares, y queremos localizar qué subclase de auxiliar concreta es en la que se delega.

ESTRUCTURA



PARTICIPANTES

- **Producto** (Documento)
  - define la interfaz de los objetos que crea el método de fabricación.
- **ProductoConcreto** (MiDocumento)
  - implementa la interfaz Producto.
- **Creador** (Aplicación)
  - declara el método de fabricación, el cual devuelve un objeto de tipo Producto. También puede definir una implementación predeterminada del método de fabricación que devuelva un objeto ProductoConcreto.
  - puede llamar al método de fabricación para crear un objeto Producto.
- **CreadorConcreto** (MiAplicacion)
  - redefine el método de fabricación para devolver una instancia de un ProductoConcreto.

COLABORACIONES

El Creador se apoya en sus subclases para definir el método de fabricación de manera que éste devuelva una instancia del ProductoConcreto apropiado.

CONSECUENCIAS

Los métodos de fabricación eliminan la necesidad de ligar clases

específicas de la aplicación a nuestro código. El código sólo trata con la interfaz Producto; además, puede funcionar con cualquier clase ProductoConcreto definida por el usuario.

Un inconveniente potencial de los métodos de fabricación es que los clientes pueden tener que heredar de la clase Creador simplemente para crear un determinado objeto ProductoConcreto. La herencia está bien cuando el cliente tiene que heredar de todos modos de la clase Creador, pero si no es así estaríamos introduciendo una nueva vía de futuros cambios.

Éstas son dos consecuencias más del patrón Factory Method:

1. *Proporciona enganches para las subclases*. Crear objetos dentro de una clase con un método de fabricación es siempre más flexible que hacerlo directamente. El Factory Method les da a las subclases un punto de enganche para proveer una versión extendida de un objeto.

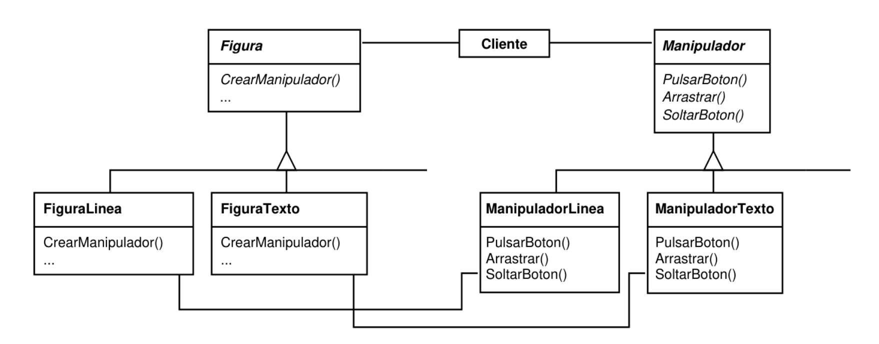
   En el ejemplo del documento, la clase Documento podría definir un método de fabricación llamado CrearDialogoDeFichero que crea un objeto de diálogo predeterminado para abrir un documento existente. Una subclase de Documento puede definir diálogos de ficheros específicos de la aplicación redefiniendo este método. En este caso el método de fabricación no es abstracto, sino que proporciona una implementación predeterminada razonable.

2. *Conecta jerarquías de clases paralelas*. En los ejemplos vistos hasta ahora, al método de fabricación sólo lo llaman los objetos Creador. Pero esto no tiene por qué ser siempre así; los clientes pueden encontrar útiles los métodos de fabricación, especialmente en el caso de jerarquías de clases paralelas.

   Las jerarquías de clases paralelas se producen cuando una clase delega alguna de sus responsabilidades a una clase separada. Pensemos en figuras gráficas que pueden manipularse interactivamente: es decir, que pueden alargarse, moverse y girarse usando el ratón. Implementar estas interacciones no siempre es fácil. Muchas veces

requiere almacenar y actualizar información que guarda el estado de la manipulación en un momento dado. Este estado es necesario sólo mientras dura la manipulación; por tanto, no necesita ser almacenado en el objeto figura. Es más, figuras diferentes se comportan de manera diferente cuando son manipuladas por el usuario. Por ejemplo, alargar una línea puede tener el efecto de mover un extremo, mientras que alargar una figura de texto puede cambiar su espaciado entre líneas.

Con estas restricciones, es mejor usar un objeto Manipulador separado que implemente la interacción y mantenga cualquier estado específico de la manipulación que sea necesario. Figuras diferentes usarán distintas subclases de Manipulador para manejar las interacciones. La jerarquía de clases de Manipulador resultante es paralela (al menos en parte) a la jerarquía de clases de Figura:



La clase Figura proporciona un método de fabricación CrearManipulador que permite que los clientes creen el Manipulador correspondiente a una figura. Las subclases de Figura redefinen este método para que devuelva una instancia de la subclase de Manipulador adecuada para cada una de ellas. Otra posibilidad es que la clase Figura implemente CrearManipulador para que devuelva una instancia predeterminada de Manipulador, y las subclases de Figura pueden simplemente heredar ese comportamiento predeterminado. Esas clases Figura no necesitan su correspondiente subclase de Manipulador —de ahí que las jerarquías sean paralelas sólo en parte—.

Es importante notar cómo el método de fabricación define la conexión entre las dos jerarquías de clases, localizando qué clases van juntas.

IMPLEMENTACIÓN

A la hora de aplicar el patrón Factory Method hemos de tener en cuenta las siguientes cuestiones:

1. *Dos variantes principales*. Las dos principales variantes del patrón Factory Method son (1) cuando la clase Creador es una clase abstracta y no proporciona una implementación para el método de fabricación que declara y (2) cuando el Creador es una clase concreta y proporciona una implementación predeterminada del método de fabricación. También es posible tener una clase abstracta que defina una implementación predeterminada, pero esto es menos común.

   El primer caso *requiere* que las subclases definan una implementación porque no hay ningún comportamiento predeterminado razonable. Esto lleva al dilema de tener que crear instancias de clases imprevisibles. En el segundo caso, el Creador concreto usa el método de fabricación principalmente por flexibilidad. Lo hace siguiendo una regla que dice: "Crear objetos en una operación aparte, para que las subclases puedan redefinir el modo en que son creados". Esta regla asegura que los diseñadores de las subclases puedan cambiar la clase de objetos que instancia su clase padre si es necesario.

2. *Métodos de fabricación parametrizados*. Otra variante del patrón permite que los métodos de fabricación creen *varios* tipos de productos. El método de fabricación recibe un parámetro que identifica el tipo de objeto a crear. Todos los objetos creados por el método compartirán la interfaz Producto. En el ejemplo del documento. Aplicación puede soportar diferentes tipos de Documentos. Pasando un parámetro extra a CrearDocumento se especifica el tipo de documento a crear. El framework de

edición gráfica Unidraw [VL90] usa este enfoque para reconstruir objetos guardados en disco. Unidraw define una clase Creator con un método de fabricación **Create** que recibe como argumento un identificador de clase. Cuando Unidraw guarda un objeto en disco, primero escribe el identificador de la clase y luego sus variables de instancia. Cuando se reconstruye el objeto desde el disco, primero lee el identificador de la clase.

Una vez que ha leído el identificador de la clase, el framework llama a Create, pasándole el identificador como el parámetro. Create busca el constructor de la clase correspondiente y lo usa para crear una instancia del objeto. Por último, Create llama a la operación Read del objeto, que lee la información restante del disco e inicializa las variables de instancia del objeto.

Un método de fabricación parametrizado tiene la siguiente forma general, donde MiProducto y TuProducto son subclases de Producto:

```
class Creador {
public:
   virtual                    Producto*
Crear(IdProducto);
};

Producto* Creador::Crear (IdProducto
id) {
   if   (id   ==   MIO)   return   new
MiProducto;
   if   (id   ==   TUYO)   return   new
TuProducto;
   //  repetir  para  los  productos
restantes…

   return 0;
}
```

Redefiniendo un método de fabricación parametrizado se puede extender o cambiar fácilmente y de manera

selectiva los productos fabricados por un Creador. Se pueden introducir nuevos identificadores para nuevos tipos de productos, o asociar identificadores existentes con productos diferentes.

Por ejemplo, una subclase de MiCreador podría cambiar MiProducto por TuProducto y admitir una nueva subclase SuProducto:

```
Producto*            MiCreador::Crear
(IdProducto id) {
     if   (id   ==   TUYO)return   new
MiProducto;
     if   (id   ==   MIO)   return   new
TuProducto;
          //   Nota:   intercambiados
TUYO y MIO

     if   (id   ==   SUYO)   return   new
SuProducto;

     return   Creador::Crear(id);   //
llamado cuando falla
                         //   todo   lo
demás
}
```

Nótese que lo último que hace esta operación es llamar al Crear de la clase padre. Eso es debido a que MiCreador::Crear sólo trata TUYO, MIO y SUYO de forma diferente a la clase padre. No está interesada en otras clases. Por tanto MiCreador *extiende* los tipos de productos creados, y delega en su padre la responsabilidad de crear todos los productos excepto unos pocos.

3. *Variantes y cuestiones específicas del lenguaje*. Lenguajes diferentes se prestan a otras interesantes variaciones.

   Los programas en Smalltalk suelen usar un método que devuelve la clase del objeto a crear. Un método de fabricación en Creador puede usar este valor para crear un producto, y un CreadorConcreto puede almacenar o

procesar dicho valor. El resultado es un enlace todavía más tardío al tipo de ProductoConcreto a crear.

Una versión en Smalltalk del ejemplo del Documento puede definir un método claseDocumento en Aplicación. El método claseDocumento devuelve la clase de Documento apropiada para crear instancias de documentos. La implementación de claseDocumento en MiAplicacion devuelve la clase MiDocumento. Así, en la clase Aplicación tenemos

```
metodoCliente
     documento        :=        self
claseDocumento new.

claseDocumento
     self subclassResponsibility
```

Y en la clase MiAplicacion tenemos

```
claseDocumento
     ^ MiDocumento
```

que devuelve la clase MiDocumento que debe ser instanciada por Aplicación.

Un enfoque aún más flexible que está estrechamente relacionado con los métodos de fabricación parametrizados es guardar la clase a crear como una variable de clase de Aplicación. De esa forma no tenemos que heredar de Aplicación para cambiar el producto.

Los métodos de fabricación en C++ son siempre funciones virtuales y a menudo virtuales puras. Hay que tener cuidado de no llamar a los métodos de fabricación en el constructor del Creador —el método de fabricación del CreadorConcreto todavía no estará disponible—.

Podemos evitar esto teniendo la precaución de acceder a

los productos sólo a través de operaciones de acceso que crean el producto cuando es necesario. En vez de crear el producto concreto en el constructor, éste simplemente lo inicializa a 0. Es el método de acceso el que devuelve el producto, pero primero comprueba que exista, y si no es así lo crea. A esta técnica se la llama a veces **inicialización perezosa**[33]. El siguiente código muestra una implementación típica:

```
class Creador {
public:
     Producto* ObtenerProducto();
protected:
     virtual                Producto*
CrearProducto();
private:
     Producto* _producto;
};

Producto*  Creador::ObtenerProducto()
{
     if (_producto == 0) {
          _producto                 =
CrearProducto();
     }
     return _producto;
}
```

4. *Usar plantillas para evitar la herencia*. Como ya hemos mencionado, otro potencial problema de los métodos de fabricación es que pueden obligar a usar la herencia sólo pata crear los o jetos Producto apropiados. Otra forma de hacer esto en C++ es proporcionar una subclase plantilla de Creador que está parametrizada con la clase de Producto:

```
class Creador {
public:
     virtual                Producto*
CrearProducto() = 0;
};
```

```
template <class ElProducto>
class    CreadorEstandar:    public
Creador {
public:
     virtual               Producto*
CrearProducto();
};

template <class ElProducto>
Producto*
CreadorEstandar<ElProducto>::CrearProducto
() {
   return new ElProducto;
}
```

Con esta plantilla, el cliente únicamente proporciona la clase del producto —no se necesita heredar de Creador—.

```
class MiProducto : public Producto {
public:
     MiProducto();
     // …
};

CreadorEstandar<MiProducto>
miCreador;
```

5. *Convenios de nominación*. Es una buena práctica usar convenios de nominación que dejen claro que estamos usando métodos de fabricación. Por ejemplo, el framework de aplicaciones para Macintosh MacApp [App89] siempre declara la operación abstracta que define el método de fabricación como Class* DoMakeClass(), donde Class es la clase Producto.

CÓDIGO DE EJEMPLO

La función CrearLaberinto construye y devuelve un laberinto. Un problema de esta función es que fija en el código la clase del laberinto y de las habitaciones, puertas y paredes. Introduciremos métodos de fabricación para permitir que las subclases elijan estos

componentes.

Primero definiremos métodos de fabricación en JuegoDelLaberinto para crear los objetos laberinto, habitación, pared y puerta:

```
class JuegoDelLaberinto {
public:
     Laberinto* CrearLaberinto();

// métodos de fabricación:

     virtual  Laberinto*  FabricarLaberinto()
const
          { return new Laberinto; }
     virtual                      Habitación*
FabricarHabitacion(int n) const
          { return new Habitacion(n); }
     virtual Pared* FabricarPared() const
          { return new Pared; }
     virtual                          Puerta*
FabricarPuerta(Habitacion* h1,
                         Habitación*      h2)
const
          { return new Habitación(hi, h2); }
};
```

Cada método de fabricación devuelve un componente laberinto de un tipo dado. JuegoDelLaberinto proporciona implementaciones predeterminadas que devuelven los tipos más simples de laberintos, habitaciones, paredes y puertas.

Ahora podemos redefinir CrearLaberinto para que use estos métodos de fabricación:

```
Laberinto*
JuegoDelLaberinto::CrearLaberinto() {
     Laberinto*         unLaberinto         =
FabricarLaberinto();
     Habitacion* h1 = FabricarHabitacion(1);
     Habitacion* h2 = FabricarHabitacion(2);
     Puerta*  laPuerta  =  FabricarPuerta(h1,
h2);

     unLaberinto->AnadirHabitacion(h1);
```

```
     unLaberinto->AnadirHabitacion(h2);

     h1->EstablecerLado(Norte,
FabricarPared());
     h1->EstablecerLado(Este, laPuerta);
     h1->EstablecerLado(Sur,
FabricarPared());
     h1->EstablecerLado(Oeste,
FabricarPared());

     h2->EstablecerLado(Norte,
FabricarPared());
     h2->EstablecerLado(Este,
FabricarPared());
     h2->EstablecerLado(Sur,
FabricarParedf));
     h2->EstablecerLado(Oeste, laPuerta);

     return unLaberinto;
}
```

Juegos diferentes pueden heredar de JuegoDelLaberinto para especializar partes del laberinto. Las subclases de JuegoDelLaberinto pueden redefinir algunos de los métodos de fabricación o todos para especificar variantes de los productos. Por ejemplo, un JuegoDelLaberintoConBombas puede redefinir los productos Habitación y Pared para devolver sus variantes con bomba:

```
class JuegoDelLaberintoConBombas : public
JuegoDelLaberinto {
public:
     JuegoDelLaberintoConBombas();

     virtual Pared* FabricarPared() const
          { return new ParedExplosionada; }

     virtual Habitacion* CrearHabitacion(int
n) const
          { return new HabitacionConBomba(n);
}
};
```

Se podría definir una variante JuegoDelLaberintoEncantado como

sigue:

```
class JuegoDelLaberintoEncantado : public
JuegoDelLaberinto {
public:
     JuegoDelLaberintoEncantado();

     virtual Habitacion*
FabricarHabitacion(int n) const
          { return new HabitacionEncantada(n,
Hechizar()); }

     virtual Puerta*
FabricarPuerta(Habitacion* h1, Habitacion*
h2) const
          { return new
PuertaQueNecesitaHechizo(h1, h2); }
protected:
     Hechizo* Hechizar() const;
};
```

USOS CONOCIDOS

Los toolkits y frameworks están plagados de métodos de fabricación. El anterior ejemplo del documento es un típico uso de MacApp y ET++ [WGM88]. El ejemplo del manipulador viene de Unidraw.

La clase View en el framework Modelo/Vista/Controlador de Smalltalk-80 tiene un método defaultController que crea un controlador, lo que puede parecer un método de fabricación [Par90]. Pero las subclases de View especifican la clase de su controlador predeterminado definiendo defaultControllerClass, que devuelve la clase a partir de la cual defaultController crea las instancias. Por tanto es defaultControllerClass quien es realmente el método de fabricación, es decir, el método que deberían redefinir las subclases.

Un ejemplo más oculto en Smalltalk-80 es el método de fabricación parserClass definido por Behavior (una superclase de todos los objetos que representan clases). Esto permite que una clase use un analizador personalizado para su código fuente. Por ejemplo, un cliente puede definir una clase SQLParser para analizar

el código fuente de una clase con sentencias SQL insertadas. La clase Behavior implementa parserClass para devolver la clase Parser estándar en Smalltalk. Una clase que incluya sentencias SQL insertadas redefine este método (como un método de clase) y devuelve la clase SQLParser.

El sistema ORB Orbix de IONA Technologies [ION94] usa el patrón Factory Method para generar un tipo adecuado de proxy (*véase* el patrón Proxy (191)) cuando un objeto solicita una referencia a un objeto remoto. El patrón Factory Method facilita reemplazar el proxy predeterminado con otro que use caché en el lado del cliente, por ejemplo.

PATRONES RELACIONADOS

El patrón Abstract Factory (79) suele implementarse con métodos de fabricación. El ejemplo que aparece en la sección Motivación del patrón Abstract Factory ilustra también un método de fabricación.

Los métodos de fabricación generalmente son llamados desde el interior de Métodos Plantilla (299). En el ejemplo del documento, NuevoDocumento es un método plantilla.

El Prototype (109) no necesita heredar de Creador. Sin embargo, suelen requerir una operación Inicializar en la clase Producto. El Creador usa Inicializar para inicializar el objeto. El patrón Factory Method no requiere dicha operación.

PROTOTYPE (Prototipo)

PROPÓSITO

Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando dicho prototipo.

MOTIVACIÓN

Podríamos construir un editor de partituras musicales adaptando un framework general para editores gráficos y añadiendo nuevos objetos que representen notas, silencios y pentagramas. El framework puede tener una paleta de herramientas para añadir estos objetos musicales a la partitura. La paleta también incluiría herramientas para seleccionar, mover y realizar otras manipulaciones de objetos musicales. Los usuarios harán clic en la herramienta de negras y la usarán para añadir negras a la partitura. O pueden usar la herramienta de movimiento para mover una nota arriba o abajo en el pentagrama, cambiando así su tono.

Supongamos que el framework proporciona una clase abstracta Gráfico para componentes gráficos, como notas y pentagramas. Además, proporcionará una clase abstracta Herramienta para definir herramientas como las de la paleta. El framework también predefine una subclase HerramientaGrafica para herramientas que crean instancias de objetos gráficos y los añaden al documento.

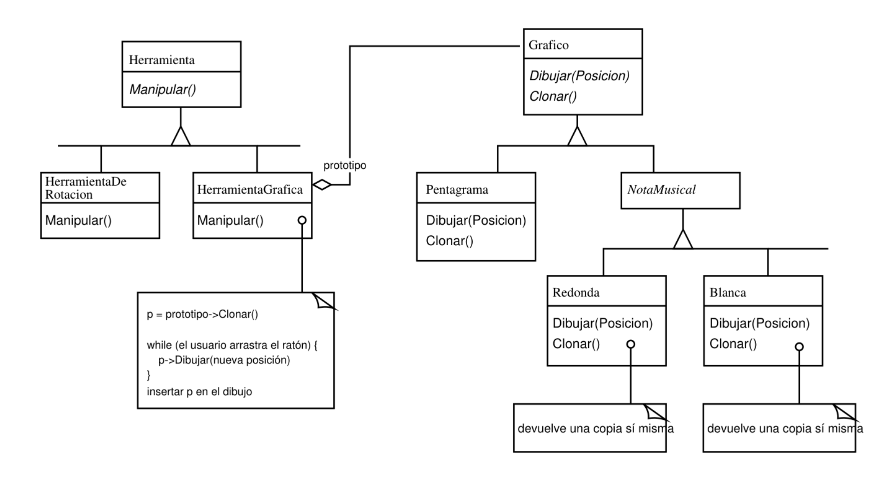
Pero HerramientaGrafica representa un problema para el diseñador del framework. Las clases para notas y pentagramas son específicas de nuestra aplicación, pero la clase HerramientaGrafica pertenece al framework, por lo que no sabe cómo crear instancias de nuestras clases musicales para añadirlas a la partitura. Podríamos crear una subclase de HerramientaGrafica para cada tipo de objeto musical, pero eso produciría muchas subclases que se diferenciarían solamente en el tipo de objeto musical que crean. Sabemos que la composición de objetos es una alternativa flexible a

la herencia. La cuestión es ¿cómo puede usarla el framework para parametrizar instancias de HerramientaGrafica con la *clase* de Gráfico que deben crear?

La solución consiste en hacer que HerramientaGrafica cree un nuevo Gráfico copiando o "clonando" una instancia de una subclase Gráfico. Llamaremos a esta instancia un **prototipo**. HerramientaGrafica está parametrizada con el prototipo que debería clonar y añadir al documento. Si todas las subclases de Gráfico admiten una operación Clonar, la HerramientaGrafica puede clonar cualquier tipo de Gráfico.

Por tanto, en nuestro editor musical, cada herramienta para crear un objeto musical es una instancia de HerramientaGrafica que está inicializada con un prototipo diferente. Cada instancia de HerramientaGrafica producirá un objeto musical clonando su prototipo y añadiendo el clon a la partitura.

Podemos usar el patrón Prototype para reducir aún más el número de clases. Tenemos clases separadas para redondas y negras, pero eso probablemente no sea necesario. En vez de eso podrían ser instancias de la misma clase inicializadas con diferentes mapas de bits y duraciones. Una herramienta para crear redondas pasa a ser simplemente una HerramientaGrafica cuyo prototipo es una NotaMusical inicializada para que sea una redonda. Esto puede reducir drásticamente el número de clases del sistema. También hace que sea más fácil añadir un nuevo tipo de nota al editor.

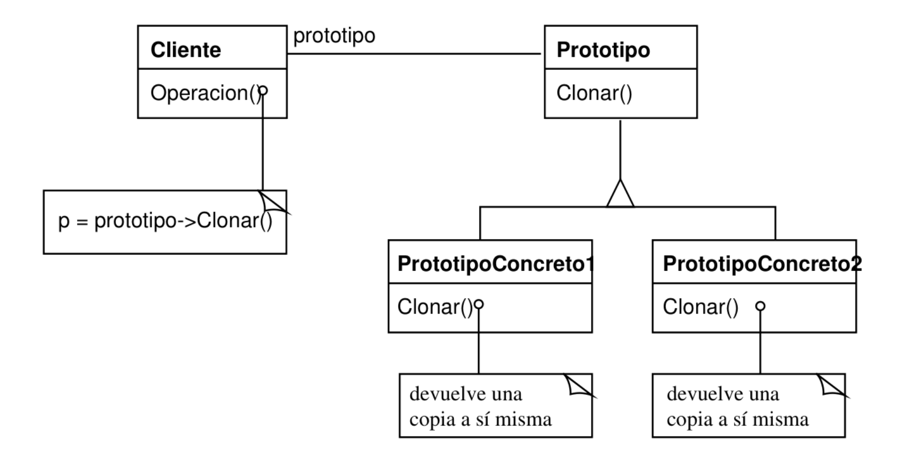


APLICABILIDAD

Úsese el patrón Prototype cuando un sistema deba ser independiente de cómo se crean, se componen y se representan sus productos; y

- cuando las clases a instanciar sean especificadas en tiempo de ejecución (por ejemplo, mediante carga dinámica); *o*
- para evitar construir una jerarquía de clases de fábricas paralela a la jerarquía de clases de los productos; *o*
- cuando las instancias de una clase puedan tener uno de entre sólo unos pocos estados diferentes. Puede ser más adecuado tener un número equivalente de prototipos y clonarlos, en vez de crear manualmente instancias de la clase cada vez con el estado apropiado.

ESTRUCTURA



PARTICIPANTES

- **Prototipo** (Gráfico)
  - declara una interfaz para clonarse.
- **PrototipoConcreto** (Pentagrama, Redonda, Blanca)
  - implementa una operación para clonarse.

- **Cliente** (HerramientaGrafica)
  - crea un nuevo objeto pidiéndole a un prototipo que se clone.

COLABORACIONES

Un cliente le pide a un prototipo que se clone.

CONSECUENCIAS

Muchas de las consecuencias del Prototype son las mismas que las de los patrones Abstract Factory (79) y Builder (89): oculta al cliente las clases producto concretas, reduciendo así el número de nombres que conocen los clientes. Además, estos patrones permiten que un cliente use clases específicas de la aplicación sin cambios.

A continuación se enumeran algunos beneficios adicionales del patrón Prototype.

1. *Añadir y eliminar productos en tiempo de ejecución*. Permite incorporar a un sistema una nueva clase concreta de producto simplemente registrando una instancia prototípica con el cliente. Esto es algo más flexible que otros patrones de creación, ya que un cliente puede instalar y eliminar prototipos en tiempo de ejecución.

2. *Especificar nuevos objetos modificando valores*. Los sistemas altamente dinámicos permiten definir comportamiento nuevo mediante la composición de objetos —por ejemplo, especificando valores para las variables de un objeto— y no definiendo nuevas clases. Podemos definir nuevos tipos de objetos creando instancias de clases existentes y registrando esas instancias como prototipos de los objetos cliente. Un cliente puede exhibir comportamiento nuevo delegando responsabilidad en su prototipo.

   Este tipo de diseño permite que los usuarios definan nuevas “clases” sin programación. De hecho, clonar un prototipo es parecido a crear una instancia de una clase. El patrón Prototype puede reducir en gran medida el número

de clases necesarias en un sistema. En nuestro editor de música, una clase HerramientaGrafica puede crear una gran variedad de objetos.

3. *Especificar nuevos objetos variando la estructura*. Muchas aplicaciones construyen objetos a partir de partes y subpartes. Los editores de diseño de circuitos, por ejemplo, construyen circuitos a partir de subcircuitos. A menudo, por comodidad, dichas aplicaciones permiten crear instancias de estructuras complejas definidas por el usuario, por ejemplo, para usar un determinado subcircuito una y otra vez.

   El patrón Prototype también permite esto. Simplemente añadimos ese subcircuito como prototipo a la paleta de circuitos disponibles. Siempre y cuando el objeto circuito compuesto implemente Clonar como una copia profunda, los circuitos con varias estructuras también pueden ser prototipos.

4. *Reduce la herencia*. El patrón Factory Method (99) suele producir una jerarquía de clases Creador que es paralela a la jerarquía de clases de productos. El patrón Prototype permite clonar un prototipo en vez de decirle a un método de fabricación que cree un nuevo objeto. Por tanto, no es en absoluto necesaria una jerarquía de clases Creador. Este beneficio es aplicable principalmente a lenguajes como C ++, que no tratan a las clases como objetos en toda regla. En los lenguajes que sí lo hacen, como Smalltalk y Objective C, resulta un beneficio menor, puesto que siempre podemos usar un objeto clase como creador. Los objetos clase ya funcionan en estos lenguajes como prototipos.

5. *Configurar dinámicamente una aplicación con clases*. Algunos entornos de tiempo de ejecución permiten cargar clases en una aplicación dinámicamente. El patrón Prototype es la clave para explotar dichas facilidades en un lenguaje como C++.

Una aplicación que quiere crear instancias de una clase cargada dinámicamente no podrá hacer referencia al constructor de ésta estáticamente. En vez de eso, el entorno en tiempo de ejecución crea automáticamente una instancia de cada clase cada vez que es cargada, y la registra con un gestor de prototipos (*véase* la Sección de Implementación). Después, la aplicación puede solicitar al gestor de prototipos instancias de nuevas clases cargadas, es decir, clases que no fueron enlazadas con el programa originalmente. El framework de aplicación ET++ [WGM88] tiene un sistema de tiempo de ejecución que usa este esquema.

El principal inconveniente del patrón Prototype es que cada subclase de Prototipo debe implementar la operación Clonar, lo cual puede ser difícil. Por ejemplo, añadir Clonar es difícil cuando las clases ya existen. Igualmente, puede ser difícil implementar Clonar cuando sus interioridades incluyen objetos que no pueden copiarse o que tienen referencias circulares.

IMPLEMENTACIÓN

El Prototype es particularmente útil con lenguajes estáticos como C ++, donde las clases no son objetos, y en los que poca o ninguna información de tipos está disponible en tiempo de ejecución. Es menos importante en lenguajes como Smalltalk y Objective C, que proporcionan lo que viene a ser un prototipo (es decir, un objeto clase) para crear instancias de cada clase.

Este patrón lo incorporan los lenguajes basados en prototipos, como Self [US87], en los cuales toda creación de objetos tiene lugar clonando un prototipo.

Hemos de tener en cuenta las siguientes cuestiones a la hora de implementar prototipos:

1. *Usar un gestor de prototipos*. Cuando el número de prototipos de un sistema no es fijo (es decir, cuando pueden crearse y destruirse dinámicamente), mantiene un registro de los prototipos disponibles. Los clientes no gestionarán ellos mismos los prototipos, sino que los

guardarán y recuperarán del registro. Un cliente le pedirá al registro un prototipo antes de clonarlo. Llamaremos a este registro un **gestor de prototipos**.

Un gestor de prototipos es un almacén asociativo que devuelve el prototipo que concuerda con una determinada clave. Tiene operaciones para registrar un prototipo con una clave y para desregistrarlo. Los clientes pueden cambiar el registro o incluso navegar por él en tiempo de ejecución. Esto permite que los clientes extiendan el sistema y lleven un inventario del mismo sin necesidad de escribir código.

2. *Implementar la operación Clonar*. La parte más complicada del patrón Prototype es implementar correctamente la operación Clonar. Es particularmente delicado cuando las estructuras de objetos contienen referencias circulares.

   La mayoría de los lenguajes proporcionan algún tipo de ayuda para clonar objetos. Por ejemplo, Smalltalk proporciona una implementación de copy que es heredada por todas las subclases de Object. C++ proporciona un constructor de copia. Pero estas facilidades no resuelven el problema de la “copia superficial frente a la copia profunda” [GR83]. Es decir, ¿clonar un objeto clona a su vez sus variables de instancia, o el objeto clonado y el original simplemente comparten sus variables?

   Una copia superficial es simple y a menudo suficiente, y es lo que Smalltalk proporciona de manera predeterminada. El constructor de copia por omisión en C++ hace una copia de miembro, lo que significa que la copia y el original compartirán sus punteros. Pero clonar prototipos con estructuras complejas normalmente requiere una copia profunda, porque el clon y el original deben ser independientes. Por tanto debemos garantizar que los componentes del clon son clones de los componentes del prototipo. La clonación nos obliga a decidir qué será compartido, si es que lo será algo.

Si los objetos del sistema proporcionan operaciones Guardar y Cargar, entonces podemos usarlas para proporcionar una implementación predeterminada de Clonar, simplemente guardando el objeto y cargándolo de nuevo a continuación. La operación Guardar guarda el objeto en un búfer en memoria, y Cargar crea un duplicado reconstruyendo el objeto a partir del búfer.

3. *Inicializar los clones*. Mientras que a algunos clientes les sirve el clon tal cual, otros querrán inicializar parte de su estado interno, o todo, con valores de su elección. Generalmente no pasamos dichos valores en la operación Clonar, porque su número variará de unas clases de prototipos a otras. Algunos prototipos pueden necesitar múltiples parámetros de inicialización; otros no necesitarán ninguno. Pasar parámetros en la operación Clonar impide tener una interfaz de clonación uniforme.

   Puede darse el caso de que nuestras clases prototipo ya definan operaciones para establecer las partes principales de su estado. Si es así, los clientes pueden usar estas operaciones inmediatamente después de la clonación. Si no, quizás tengamos que introducir una operación Inicializar (*véase* la Sección Código de Ejemplo) que tome parámetros de inicialización como argumentos y establezca el estado interno del clon. Hemos de tener cuidado con las operaciones Clonar que hacen una copia profunda —quizá tengamos que borrar las copias (ya sea explícitamente o desde dentro de Inicializar) antes de reinicializarlas—.

CÓDIGO DE EJEMPLO

Definiremos FabricaDePrototiposLaberinto como subclase de la clase FabricaDeLaberintos (página 83). FabricaDePrototiposLaberinto será inicializada con prototipos de los objetos que creará, de manera que no habrá que heredar de ella sólo para cambiar las clases de paredes o habitaciones que crea.

FabricaDePrototiposLaberinto aumenta la interfaz FabricaDeLaberintos con un constructor que toma como argumentos

los prototipos:

```
class  FabricaDePrototiposLaberinto  :  public
FabricaDeLaberintos {
public:
     FabricaDePrototiposLaberinto(Laberinto*,
Pared*, Habitacion*, Puerta*);

     virtual   Laberinto*   HacerLaberinto()
const;
     virtual Habitacion* HacerHabitacion(int)
const;
     virtual Pared* HacerPared() const;
     virtual Puerta* HacerPuerta(Habitacion*,
Habitacion*) const;

private:
     Laberinto* _prototipoLaberinto;
     Habitacion* _prototipoHabitacion;
     Pared* prototipoPared;
     Puerta* _prototipoPuerta;
   };
```

El nuevo constructor simplemente inicializa sus prototipos:

```
FabricaDePrototiposLaberinto::FabricaDePrototiposLaberinto
(
     Laberinto* l, Pared* p, Habitacion* h,
Puerta* pu
) {
     _prototipoLaberinto = l;
     _prototipoPared = p;
     _prototipoHabitacion = h;
     _prototipoPuerta = pu;
}
```

Las funciones miembro para crear paredes, habitaciones y puertas son parecidas: cada una clona un prototipo y luego lo inicializa. A continuación se presentan las definiciones de HacerPared y HacerPuerta:

```
Pared*
FabricaDePrototiposLaberinto::HacerPared   ()
const {
     return _prototipoPared->Clonar();
}

Puerta*
FabricaDePrototiposLaberinto::HacerPuerta (
          Habitacion*  h1,  Habitacion  *h2)
const {
     Puerta*  puerta  =  _prototipoPuerta-
>Clonar();
     puerta->Inicializar(h1, h2);
     return puerta;
}
```

Podemos usar FabricaDePrototiposLaberinto para crear un Laberinto prototípico o predeterminado simplemente inicializándolo con prototipos de los componentes básicos de Laberinto:

```
JuegoDelLaberinto juego;
FabricaDePrototiposLaberinto
FabricaDeLaberintosSimples(
     new   Laberinto,   new   Pared,   new
Habitacion, new Puerta
);

Laberinto*             Laberinto            =
juego.CrearLaberinto(FabricaDeLaberintosSimples);
```

Para cambiar el tipo de Laberinto, inicializamos FabricaDePrototiposLaberinto con un conjunto diferente de prototipos. La siguiente llamada crea un Laberinto con una puerta PuertaExplosionada y una HabitacionConBomba:

```
FabricaDePrototiposLaberinto
FabricaDeLaberintosConBomba(
     new Laberinto, new ParedExplosionada,
     new HabitacionConBomba, new Puerta
);
```

Un objeto que puede ser usado como prototipo, como por ejemplo una instancia de Pared, debe admitir la operación Clonar. También tiene que tener un constructor de copia para la clonación. Puede necesitar también otra operación para reinicializar su estado interno. Añadiremos la operación Inicializar a Puerta para que los clientes puedan inicializar los clones de Habitación.

Compárese la siguiente definición de Puerta con la de la página 76:

```
class Puerta : public LugarDelMapa {
public:
     Puerta();
     Puerta(const Puerta&);

     virtual  void  Inicializar(Habitacion*,
Habitacion*);
     virtual Puerta* Clonar() const;

     virtual void Entrar();
     Habitacion* OtroLadoDe(Habitacion*);
private:
     Habitacion* _habitacion1;
     Habitacion* _habitacion2;
};

Puerta::Puerta (const Puerta& otra) {
     _habitacion1 = otra._habitacion1;
     _habitacion2 = otra._habitacion2;
}

void  Puerta::Inicializar  (Habitación*  h1,
Habitación* h2) {
     _habitacion1 = h1;
     _habitacion2 = h2;
}

Puerta* Puerta::Clonar () const {
     return new Puerta(*this);
}
```

La subclase ParedExplosionada debe redefinir Clonar e implementar su correspondiente constructor de copia.

```
class ParedExplosionada : public Pared {
public:
     ParedExplosionada();
     ParedExplosionada(const
ParedExplosionada&);

     virtual Pared* Clonar() const;
     bool TieneBomba();
private:
     bool _bomba;
};

ParedExplosionada::ParedExplosionada   (const
ParedExplosionada&
                         otra)  :  Pared(otra)
{
     _bomba = otra._bomba;
}

Pared* ParedExplosionada::Clonar () const {
     return new ParedExplosionada(*this);
}
```

Aunque ParedExplosionada::Clonar devuelve Pared*, su implementación devuelve un puntero a una nueva instancia de una subclase, es decir, ParedExplosionada*. Definimos Clonar de ese modo en la clase base para asegurar que los clientes que clonan el prototipo no tienen que conocer a sus subclases concretas. Los clientes nunca deberían necesitar convertir el valor devuelto por Clonar al tipo deseado.

En Smalltalk podemos reutilizar el método estándar copy heredado de Object para clonar cualquier LugarDelMapa. Podemos usar FabricaDeLaberintos para producir los prototipos necesarios; por ejemplo, podemos crear una Habitación suministrando el nombre #Habitación. La FabricaDeLaberintos tiene un diccionario que hace corresponder nombres con prototipos. Su método hacer: se parece a esto:

```
hacer: nombreParte
     ^  (catalogoDePartes  at:  nombreParte)
copy
```

Dados los métodos apropiados para inicializar la FabricaDeLaberintos con prototipos, podríamos crear un Laberinto simple con el código siguiente:

```
CrearLaberinto
     on: (FabricaDeLaberintos new
          with: Puerta new named: #puerta;
          with: Pared new named: Spared;
          with:    Habitación    new    named:
#habitación;
          yourself)
```

donde la definición del método de clase on: para CrearLaberinto sería

```
on: unaFabrica
     ¦ Habitacion1 Habitacion2 ¦
     Habitacion1    :=    (unaFabrica    hacer:
#habitacion)
                       ubicación: 1@1.
     Habitacion2    :=    (unaFabrica    hacer:
#habitacion)
                       ubicación: 2@1.
     puerta  :=  (unaFabrica  hacer:  #puerta)
de: habitacion1
                          a: habitacion2.

     Habitacion1
          lado:   #norte   put:   (unafabrica
hacer: #pared);
          lado: #este put: puerta;
          lado: #sur put: (unafabrica hacer:
#pared);
          lado:   #oeste   put:   (unafabrica
hacer: #pared).
     Habitacion2
          lado:   #norte   put:   (unafabrica
hacer: #pared);
          lado:   ((este   put:   (unafabrica
hacer: #pared);
          lado: #sur put: (unafabrica hacer:
#pared);
          lado: #oeste put: puerta.
```

```
^ Laberinto new
    anadirHabitacion: habitacion1;
    anadirHabitacion: habitacion2;
    yourself
```

USOS CONOCIDOS

Quizás el primer ejemplo del patrón Prototipo fue en el sistema Sketchpad de Ivan Sutherland [Sut63]. La primera aplicación ampliamente conocida del patrón en un lenguaje orientado a objetos fue en ThingLab, donde los usuarios podían formar un objeto compuesto y luego promocionarlo a un prototipo instalándolo en una biblioteca de objetos reutilizables [Bor81]. Goldberg y Robson mencionan prototipos como patrón [GR83], pero Coplien [Cop92] da una descripción mucho más completa, incluyendo idiomas para C++ relacionados con el patrón Prototype, con muchos ejemplos y variaciones.

Etgdb es el *front-end* de un depurador basado en ET++ que proporciona una interfaz basada en ratón para diferentes depuradores de líneas. Cada depurador tiene su correspondiente subclase DebuggerAdaptor. Por ejemplo, GdbAdaptor adapta etgdb a la sintaxis de las órdenes de GNU gdb, mientras que SunDbxAdaptor adapta etgdb al depurador dbx de Sun. Etgdb no tiene una serie de clases DebuggerAdaptor incrustadas en el código, sino que lee el nombre del adaptador a usar de una variable de entorno, busca un prototipo con el nombre especificado en la tabla global y luego clona el prototipo. Se pueden añadir nuevos prototipos a etgdb enlazándolos con el DebuggerAdaptor que funcione con ese depurador.

La "biblioteca de técnicas de interacción" en Mode Composer almacena prototipos de objetos que soportan varias técnicas de interacción [Sha90]. Cualquier técnica de interacción creada por el Mode Composer puede usarse como prototipo poniéndola en esta biblioteca. El patrón Prototype permite que Mode Composer soporte un conjunto ilimitado de técnicas de interacción.

El ejemplo del editor musical examinado anteriormente está basado en el framework de dibujo Unidraw [VL90].

PATRONES RELACIONADOS

Prototype y Abstract Factory (79) son patrones rivales en algunos aspectos, como se analiza al final de este capítulo. No obstante, también pueden usarse juntos. Una fábrica abstracta puede almacenar un conjunto de prototipos a partir de los cuales clonar y devolver objetos producto.

Los diseños que hacen un uso intensivo de los patrones Composite (151) y Decorator (161) suelen beneficiarse también del Prototype.

SINGLETON (Único)

PROPÓSITO

Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.

MOTIVACIÓN

Es importante que algunas clases tengan exactamente una instancia. Aunque puede haber muchas impresoras en un sistema, sólo debería haber una cola de impresión. Igualmente, sólo debería haber un sistema de ficheros y un gestor de ventanas. Un filtro digital tendrá un convertidor A/D. Un sistema de contabilidad estará dedicado a servir a una única compañía.

¿Cómo podemos asegurar que una clase tenga una única instancia y que ésta sea fácilmente accesible? Una variable global hace accesible a un objeto, pero no nos previene de crear múltiples instancias de objetos.

Una solución mejor es hacer que sea la propia clase la responsable de su única instancia. La clase puede garantizar que no se puede crear ninguna otra instancia (interceptando las peticiones para crear nuevos objetos), y puede proporcionar un modo de acceder a la instancia. En eso consiste el patrón Singleton.

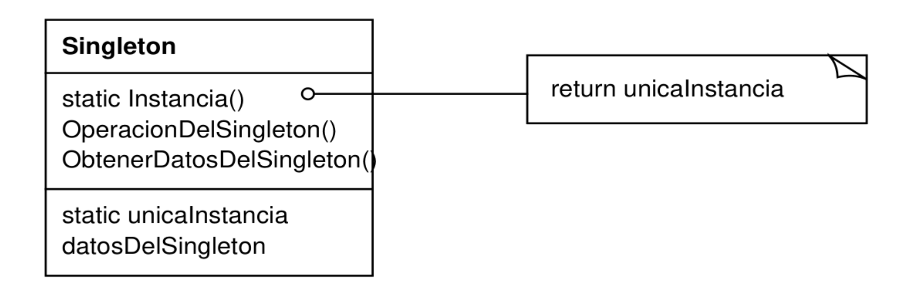
APLICABILIDAD

Usaremos el patrón Singleton cuando

- deba haber exactamente una instancia de una clase, y ésta debe ser accesible a los clientes desde un punto de acceso conocido.
- la única instancia debería ser extensible mediante herencia, y los clientes deberían ser capaces de usar una instancia

extendida sin modificar su código.

ESTRUCTURA



PARTICIPANTES

- **Singleton**
  - define una operación Instancia que permite que los clientes accedan a su única instancia. Instancia es una operación de clase (es decir, un método de clase en Smalltalk y una función miembro estática en C++).
  - puede ser responsable de crear su única instancia.

COLABORACIONES

Los clientes acceden a la instancia de un Singleton exclusivamente a través de la operación Instancia de éste.

CONSECUENCIAS

El patrón Singleton proporciona varios beneficios:

1. *Acceso controlado a la única instancia*. Puesto que la clase Singleton encapsula su única instancia, puede tener un control estricto sobre cómo y cuándo acceden a ella los clientes.
2. *Espacio de nombres reducido*. El patrón Singleton es una mejora sobre las variables globales. Evita contaminar el espacio de nombres con variables globales que almacenen las instancias.

3. *Permite el refinamiento de operaciones y la representación*. Se puede crear una subclase de la clase Singleton, y es fácil configurar una aplicación con una instancia de esta clase extendida. Podemos configurar la aplicación con una instancia de la clase necesaria en tiempo de ejecución.
4. *Permite un número variable de instancias*. El patrón hace que sea fácil cambiar de opinión y permitir más de una instancia de la clase Singleton. Además, podemos usar el mismo enfoque para controlar el número de instancias que usa la aplicación. Sólo se necesitaría cambiar la operación que otorga acceso a la instancia del Singleton.
5. *Más flexible que las operaciones de clase*. Otra forma de empaquetar la funcionalidad de un Singleton es usar operaciones de clase (es decir, funciones miembro estáticas en C++ o métodos de clase en Smalltalk). Pero ambas técnicas de estos lenguajes dificultan cambiar un diseño para permitir más de una instancia de una clase. Además, las funciones miembro estáticas en C++ nunca son virtuales, por lo que las subclases no las pueden redefinir polimórficamente.

IMPLEMENTACIÓN

A continuación se presentan unas cuestiones de implementación a tener en cuenta a la hora de usar el patrón Singleton:

1. *Garantizar una única instancia*. El patrón Singleton hace que la única instancia sea una instancia normal de la clase, pero dicha clase se escribe de forma que sólo se pueda crear una instancia. Una forma usual de hacer esto es ocultar la operación que crea la instancia tras una operación de clase (es decir, una función miembro estática o un método de clase) que garantice que sólo se crea una única instancia. Esta operación tiene acceso a la variable que contiene la instancia, y se asegura de que la variable está inicializada con dicha instancia antes de devolver su valor. Este enfoque garantiza que un Singleton se cree e inicialice antes de su primer uso.

Podemos definir la operación de clase en C++ mediante una función miembro estática Instancia de la clase Singleton. Singleton también define una variable miembro estática instancia que contiene un puntero a su única instancia.

La clase Singleton se declara como

```
class Singleton {
public:
     static Singleton* Instancia();
protected:
     Singleton();
private:
     static Singleton* _instancia;
};
```

La correspondiente implementación es

```
Singleton*  Singleton::_instancia  =
0;

Singleton* Singleton::Instancia !) {
     if (_instancia == 0) {
          _instancia      =      new
Singleton;
     }
     return _instancia;
}
```

Los clientes acceden al Singleton exclusivamente a través de la función miembro Instancia. La variable _instancia se inicializa a 0, y la función miembro estática Instancia devuelve su valor, inicializándola con la única instancia en caso de que sea 0. Instancia usa inicialización perezosa; el valor que devuelve no se crea y se almacena hasta que se accede a él por primera vez.

Nótese que el constructor se declara como protegido. Un cliente que trate de crear una instancia de Singleton

directamente obtendrá un error en tiempo de compilación. Esto garantiza que sólo se puede crear una instancia.

Además, puesto que _instancia es un puntero a un objeto Singleton, la función miembro Instancia puede asignar a esta variable un puntero a una subclase de Singleton. Daremos un ejemplo de esto en el Código de Ejemplo.

Hay otra cosa en que fijarse en la implementación C++. No basta con definir el Singleton como un objeto global o estático y luego confiar en la inicialización automática. Hay tres razones para ello:

1. No podemos garantizar que sólo de declarará una única instancia de un objeto estático.
2. Podríamos no tener la información necesaria para crear una instancia de cada Singleton en el momento de la inicialización estática. Un Singleton puede necesitar valores que se calculan después en la ejecución del programa.
3. C++ no define el orden en que se llama a los constructores de objetos globales cuando hay varias unidades de traducción [ES90]. Esto significa que no pueden existir dependencias entre objetos Singleton; si hay alguna, es inevitable que se produzcan errores.

Una desventaja (si bien es cierto que pequeña) añadida al enfoque de un objeto global o estático es que obliga a que se creen todos los objetos Singleton, independientemente de que se usen o no. Usar una función miembro estática evita todos estos problemas.

En Smalltalk, la función que devuelve la única instancia se implementa como un método de clase de la clase Singleton. Para asegurar que sólo se crea una instancia, redefine la operación new. La clase Singleton resultante podría tener los siguientes dos métodos de clase, donde UnicaInstancia es una variable de clase que no se usa en ningún otro sitio:

```
new
     self error: 'no se puede crear
un nuevo objeto'

default
     UnicaInstancia  isNil  ifTrue:
[UnicaInstancia : = super new].
     ^ UnicaInstancia
```

2. *Crear una subclase de Singleton*. El principal problema no es definir la subclase sino instalar su única instancia de manera que los clientes la puedan usar. En esencia, la variable que hace referencia a la única instancia debe ser inicializada con una instancia de la subclase. La técnica más sencilla es determinar qué Singleton queremos usar en la operación Instancia de Singleton. Uno de los ejemplos del Código de Ejemplo muestra cómo implementar esta técnica con variables de entorno.

   Otro modo de elegir la subclase de Singleton es sacar la implementación de Instancia fuera de la clase padre (por ejemplo. FabricaDeLaberintos) y ponerla en la subclase. Eso permite que un programador de C+ + decida la clase del Singleton durante el enlazado (por ejemplo, enlazándola con un fichero objeto que contenga una implementación diferente) pero la oculta a los clientes del Singleton.

   Este enfoque fija la elección de la clase del Singleton durante el enlazado, lo que dificulta elegir la clase del Singleton en tiempo de ejecución. Usar instrucciones condicionales para determinar la subclase es más flexible, pero fija el conjunto de posibles clases Singleton. Ninguno de los dos enfoques es lo suficientemente flexible para todos los casos.

   Un enfoque más flexible usa un **registro de objetos Singleton**. En vez de que sea Instancia quien defina el conjunto de posibles clases Singleton, las clases Singleton pueden registrar su única instancia por su nombre en un

registro que sea conocido por todos.

El registro establece una correspondencia entre nombres y objetos Singleton. Cuando Instancia necesita un Singleton, consulta el registro, pidiendo un Singleton por su nombre. El registro busca el Singleton correspondiente (si existe) y lo devuelve. Este enfoque fibra a Instancia de tener que conocer todas las posibles clases o instancias de Singleton. Todo lo que necesita es una interfaz común para todas las clases Singleton que incluya operaciones para el registro:

```
class Singleton {
public:
     static   void   Registrar(const
char* nombre, Singleton*);
     static Singleton* Instancia();
protected:
     static  Singleton*  Buscar(const
char* nombre);
private:
     static Singleton* _instancia;
     static
List<ParNombreSingleton>* registro;
};
```

Registrar registra la instancia del Singleton con el nombre especificado. Para mantener el registro simple, haremos que guarde una lista de objetos ParNombreSingleton. Cada ParNombreSingleton hace corresponder un nombre con un Singleton. La operación Buscar busca un Singleton dado su nombre. Supondremos que una variable de entorno especifica el nombre del Singleton que se desea.

```
Singleton* Singleton::Instancia () {
     if (_instancia == 0) {
          const                 char*
nombreSingleton                      =
getenv("SINGLETON");
          //  proporcionada  por  el
usuario o el entorno al principio
```

```
            _instancia                =
Buscar(nombreSingleton);
            // Buscar devuelve 0 si no
existe tal Singleton
      }
      return _instancia;
}
```

¿Dónde se registran las propias clases Singleton? Una posibilidad es en su constructor. Por ejemplo, una subclase MiSingleton podría hacer lo siguiente:

```
MiSingleton::MiSingleton() {
      // …
      Singleton::Registrar("MiSingleton",
this);
}
```

Por supuesto, el constructor no será llamado al menos que alguien cree una instancia de la clase, ¡lo que reproduce el problema que el patrón Singleton está tratando de resolver! Podemos evitar este problema en C++ definiendo una instancia estática de MiSingleton. Por ejemplo, podemos definir

```
static MiSingleton elSingleton;
```

en el fichero que contiene la implementación de MiSingleton.

La clase Singleton ya no es la responsable de crear el Singleton. En vez de eso, su principal responsabilidad es hacer que en el sistema se pueda elegir el Singleton y acceder a él. El enfoque del objeto estático todavía tiene un inconveniente potencial: hay que crear las instancias de todas las posibles subclases de Singleton o, si no, no se registrarán.

CÓDIGO DE EJEMPLO

Supongamos que definimos una clase FabricaDeLaberintos para

construir laberintos, como se describió en la página 83. La clase FabricaDeLaberintos define una interfaz para construir las diferentes partes de un laberinto. Las subclases pueden redefinir sus operaciones para devolver instancias de clases producto especializadas, como objetos ParedExplosionada en vez de simples objetos Pared.

Lo relevante aquí es que la aplicación Laberinto sólo necesita una instancia de una fábrica de laberintos, y esa instancia debería estar disponible para el código que construye cualquier parte del laberinto. Aquí es donde entra en juego el patrón Singleton. Al hacer a la FabricaDeLaberintos un Singleton, hacemos que el objeto laberinto sea accesible globalmente sin depender de variables globales.

Para simplificar, supondremos que nunca crearemos subclases de FabricaDeLaberintos (enseguida veremos cómo sería en ese caso). En C++ hacemos que sea una clase Singleton, añadiendo una operación estática Instancia y un miembro también estático _instancia, la que guarde la única instancia. También debemos declarar al constructor como protegido para prevenir la creación de instancias accidentalmente, lo que podría dar lugar a más de una instancia.

```
class FabricaDeLaberintos {
public:
     static FabricaDeLaberintos* Instancia();

     // aquí iría la interfaz existente
protected:
     FabricaDeLaberintos();
private:
     static FabricaDeLaberintos* _instancia;
};
```

La correspondiente implementación es

```
FabricaDeLaberintos*
FabricaDeLaberintos::_instancia = 0;

FabricaDeLaberintos*
```

```
FabricaDeLaberintos::Instancia () {
     if (_instancia == 0) {
          _instancia          =          new
FabricaDeLaberintos;
     }
     return _instancia;
}
```

Pensemos ahora qué ocurre cuando hay subclases de FabricaDeLaberintos y la aplicación tiene que decidir cuál usar. Seleccionaremos el tipo de laberinto por medio de una variable de entorno y añadiremos código que cree una instancia de la subclase apropiada de FabricaDeLaberintos, basándose en el valor de la variable de entorno. La operación Instancia es un buen lugar donde poner este código, dado que ya crea una instancia de FabricaDeLaberintos:

```
FabricaDeLaberintos*
FabricaDeLaberintos::Instancia () {
     if (_instancia == 0) {
          const   char*   estiloLaberinto   =
getenv("ESTILOLABERINTO");

          if          (strcmplestiloLaberinto,
"conBombas") == 0) {
               _instancia        =        new
FabricaDeLaberintosConBombas;

          } else if (strcmp(estiloLaberinto,
"encantado") == 0) {
               _instancia        =        new
FabricaDeLaberintosEncantados;

          // … otras posibles subclases

          } else {// por omisión
               _instancia        =        new
FabricaDeLaberintos;
          }
     }
     return _instancia;
}
```

Nótese que debemos modificar Instancia cada vez que definimos una nueva subclase de FabricaDeLaberintos. Eso puede que no sea un problema en esta aplicación, pero podría serlo para fábricas abstractas definidas en un framework.

Una posible solución sería usar el enfoque del registro descrito en la sección Implementación. También podría ser útil el enlazado dinámico —previene a la aplicación de tener que cargar todas las subclases que no se usan.

USOS CONOCIDOS

Un ejemplo del patrón Singleton en Smalltalk-80 [Par90] es el conjunto de cambios al código, ChangeSet current. Un ejemplo más sutil es la relación entre las clases y sus metaclases. Una metaclase es la clase de una clase, y cada metaclase tiene una instancia. Las metaclases no tienen nombres (salvo indirectamente, a través de su única instancia), pero conocen a su única instancia y normalmente no crearán otra.

El toolkit de interfaces de usuario Interviews [LCI+92] usa el patrón Singleton para acceder a la única instancia de sus clases Session y WidgetKit, entre otras. La clase Session define el bucle principal de despacho de eventos de la aplicación, almacena la base de datos con las preferencias estilísticas del usuario y gestiona las conexiones a uno o más dispositivos físicos de visualización. La clase WidgetKit es una fábrica abstracta para definir el aspecto y el comportamiento de los elementos de la interfaz de usuario. La operación WidgetKit::instance() determina la subclase concreta de WidgetKit de la que se va a crear una instancia basándose en una variable de entorno definida por la clase Session. Una operación parecida en Session determina si se soportan monitores monocromo o en color, y configura según eso la única instancia de Session.

PATRONES RELACIONADOS

Hay muchos patrones que pueden implementarse usando el patrón Singleton. Véanse el Abstract Factory (79), el Builder (89) y el Prototype (109).

DISCUSIÓN SOBRE LOS PATRONES DE CREACIÓN

Hay dos formas habituales de parametrizar un sistema con las clases de objetos que crea. Una es heredar de la clase que crea los objetos; esto se corresponde con el uso del patrón Factory Method (99). HI principal inconveniente de este enfoque es que puede requerir crear una nueva subclase simplemente para cambiar la clase del producto. Dichos cambios pueden tener lugar en cascada. Así, por ejemplo, cuando el creador del producto es a su vez creado por un método de fabricación, habrá que redefinir también a su creador.

El otro modo de parametrizar un sistema se basa más en la composición de objetos; consiste en definir un objeto que sea responsable de conocer la clase de los objetos producto, y hacer que sea un parámetro del sistema. Éste es un aspecto clave de los patrones Abstract Factory (79), Builder (89) y Prototype (109). Los tres implican crear un nuevo "objeto fábrica" cuya responsabilidad es crear objetos producto. El objeto fábrica del Abstract Factory produce objetos de varias clases, mientras que en el caso del Builder construye un producto complejo incrementalmente usando un protocolo de complejidad similar. El Prototype, por su parte, hace que su objeto fábrica construya un producto copiando un objeto prototípico. En este caso, el objeto fábrica y el prototipo son el mismo objeto, ya que el prototipo es el responsable de devolver el producto.

Pensemos en el framework de editores de dibujo descrito en el patrón Prototype. Hay varias formas de parametrizar una HerramientaGrafica con la clase del producto:

- Si aplicamos el patrón Factory Method, se creará en la paleta una subclase de HerramientaGrafica para cada subclase de Gráfico. HerramientaGrafica tendrá una operación NuevoGrafico que será redefinida por cada una de sus subclases.

- Si aplicamos el patrón Abstract Factory, habrá una jerarquía de clases de FabricaDeGraficos, una para cada subclase de Gráfico. En este caso, cada fábrica crea un solo producto: FabricaDeCirculos creará Circulos, FabricaDeLineas creará Lineas, y así sucesivamente. Una HerramientaGrafica estará parametrizada con una fábrica para crear el tipo apropiado de Gráfico.
- Si aplicamos el patrón Prototype, cada subclase de Gráfico implementará la operación Clonar, y cada HerramientaGrafica estará parametrizada con un prototipo del Gráfico que crea.

Qué patrón es mejor depende de muchos factores. En nuestro framework de editores de dibujo, el patrón Factory Method es el más fácil de usar al principio. Resulta sencillo definir una nueva subclase de HerramientaGrafica, y las instancias de HerramientaGrafica se crean sólo cuando se define la paleta. El principal inconveniente de esta opción es que proliferarán las subclases de HerramientaGrafica, y ninguna de ellas hará gran cosa.

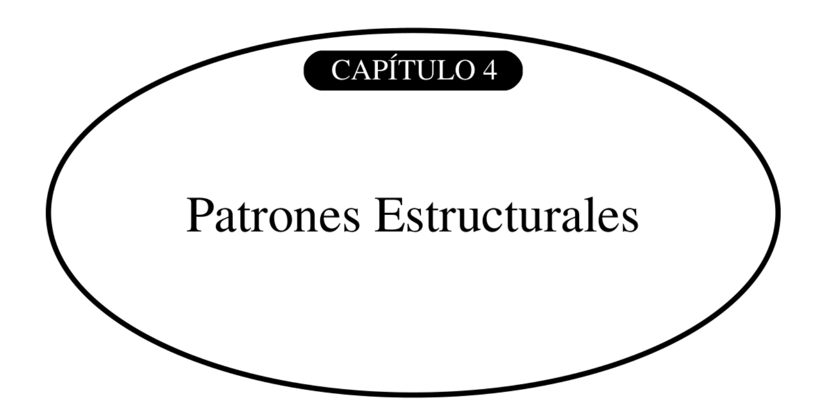
El Abstract Factory no ofrece mucha mejora, ya que requiere una jerarquía de clases FabricaDeGraficos igualmente grande. El Abstract Factory sería preferible al Factory Method sólo si ya hubiera una jerarquía de clases FabricaDeGraficos, ya sea porque la proporcione el compilador de manera automática (como en Smalltalk o en Objective C) o porque sea necesaria en otra parte del sistema.

De todos ellos, el patrón Prototype es probablemente el mejor para el framework de editores de dibujo, ya que sólo requiere implementar una operación Clonar en cada clase de Gráfico. Esto reduce el número de clases, y la operación Clonar puede usarse para otros propósitos además de la mera creación de instancias (por ejemplo, para una operación de menú Duplicar).

El Factory Method hace que un diseño sea más adaptable a cambio de sólo un poco más de complejidad. Otros patrones de diseño requieren clases nuevas, mientras que el Factory Method sólo necesita una nueva operación. La gente suele usar cl Factory Method como la forma estándar de creación de objetos, pero no es necesario cuando la clase de la que se crean instancias no cambia nunca o cuando la creación de instancias tiene lugar en una

operación que puede ser redefinida fácilmente por las subclases, como en una operación de inicialización.

Los diseños que usan los patrones Abstract Factory, Prototype o Builder son todavía más flexibles que aquellos que usan el Factory Method, pero también son más complejos. Muchas veces, los diseños empiezan usando el Factory Method y luego evolucionan hacia los otros patrones de creación a medida que el diseñador descubre dónde es necesaria más flexibilidad. Conocer muchos patrones de diseño nos da más posibilidades de elección cuando estamos analizando los pros y los contras de un criterio de diseño frente a otro.



CONTENIDO DEL CAPÍTULO

Los patrones estructurales se ocupan de cómo se combinan las clases y los objetos para formar estructuras más grandes. Los patrones estructurales *de clases* hacen uso de la herencia para componer interfaces o implementaciones. A modo de ejemplo sencillo, pensemos en cómo la herencia múltiple mezcla dos o más clases en una sola. El resultado es una clase que combina las propiedades de sus clases padre. Este patrón es particularmente útil para lograr que funcionen juntas bibliotecas de clases desarrolladas de forma independiente. Otro ejemplo es la versión de clases del patrón Adapter (131). En general, un adaptador hace que una interfaz (la de la clase adaptable) se ajuste a otra, proporcionando así una abstracción uniforme de interfaces diferentes. La forma de lograr esto en un adaptador de clases es heredando privadamente de una clase adaptable. A continuación el adaptador expresa su interfaz en términos de la de la clase adaptable.

En vez de combinar interfaces o implementaciones, los patrones estructurales *de objetos* describen formas de componer objetos para obtener nueva funcionalidad. La flexibilidad añadida de la composición de objetos viene dada por la capacidad de cambiar la composición en tiempo de ejecución, lo que es imposible con la composición de clases estática.

El Composite (151) es un ejemplo de patrón estructural de objetos. Describe cómo construir una jerarquía de clases formada por dos tipos de objetos: primitivos y compuestos. Los objetos compuestos permiten combinar objetos primitivos, así como otros objetos compuestos, para formar estructuras todo lo complejas que se quiera. En el patrón Proxy (191) un objeto proxy actúa como un sustituto o un representante de otro objeto. Se puede usar un proxy de muchas formas. Puede hacer de representante local de un objeto que está en un espacio de direcciones remoto. Puede representar un objeto más grande que debería ser cargado a petición. Puede proteger el acceso a un objeto confidencial. Los *proxies*

proporcionan un nivel de indirección a determinadas propiedades de los objetos. Por tanto, pueden restringir, aumentar o alterar dichas propiedades.

El patrón Flyweight (179) define una estructura para compartir objetos. Los objetos se comparten por al menos dos razones: eficiencia y consistencia. Este patrón se basa en el compartimiento como mecanismo para lograr un uso más eficiente del espacio. Las aplicaciones que usan muchos objetos deben prestar especial atención a los costes de cada objeto. Se pueden lograr ahorros sustanciales compartiendo objetos en vez de duplicarlos. Pero los objetos pueden compartirse sólo si no tienen un estado que dependa del contexto. Los objetos *flyweight* (peso ligero) no tienen dicho estado. Cualquier información adicional que necesiten para llevar a cabo su tarea se les pasa en el momento que sea necesaria. Así, sin estado dependiente del contexto, los objetos "peso ligero" pueden compartirse sin problemas.

Mientras que el patrón Flyweight muestra cómo crear muchos objetos pequeños, el patrón Facade (171) muestra cómo hacer que un único objeto represente un subsistema completo. Una fachada es un representante de un conjunto de objetos. La fachada lleva a cabo sus responsabilidades reenviando mensajes a los objetos que representa. El patrón Bridge (141) separa la abstracción de un objeto de su implementación, de manera que ambas puedan ser modificadas por separado.

El patrón Decorator (161) describe cómo añadir responsabilidades a objetos dinámicamente. Es un patrón estructural que compone objetos recursivamente para permitir un número ilimitado de responsabilidades adicionales. Por ejemplo, un objeto Decorador que contiene un componente de interfaz de usuario puede añadir un adorno, como un borde o un sombreado, al componente, o bien añadir funcionalidad, como capacidad de desplazamiento o *zoom*. Podemos añadir dos adornos simplemente anidando un objeto Decorador dentro de otro, v así sucesivamente para adornos adicionales. Para llevar a cabo esto, cada objeto Decorador debe ajustarse a la interfaz de su componente y reenviarle los mensajes. El Decorador puede hacer su trabajo (como dibujar un borde alrededor del componente) antes o después de reenviar un mensaje.

Muchos patrones estructurales están de alguna manera relacionados. Trataremos dichas relaciones al final del capítulo.

ADAPTER
(Adaptador)

PROPÓSITO

Convierte la interfaz de una clase en otra interfaz que es la que esperan los clientes. Permite que cooperen clases que de otra forma no podrían por tener interfaces incompatibles.

TAMBIÉN CONOCIDO COMO

Wrapper (Envoltorio)

MOTIVACIÓN

A veces una clase de un toolkit que ha sido diseñada para reutilizarse, no puede hacerlo porque su interfaz no coincide con la interfaz específica del dominio que requiere la aplicación.

Pensemos, por ejemplo, en un editor de dibujo que permita que los usuarios dibujen y ubiquen elementos gráficos (líneas, polígonos, texto, etc.) en dibujos y diagramas. La abstracción fundamental del editor de dibujo es el objeto gráfico, que tiene una forma modificable y que puede dibujarse a sí mismo. La interfaz de los objetos gráficos está definida por una clase abstracta llamada Forma. El editor define una subclase de Forma para cada tipo de objeto gráfico: una clase FormaLinea para las líneas, otra FormaPoligono para los polígonos, etcétera.

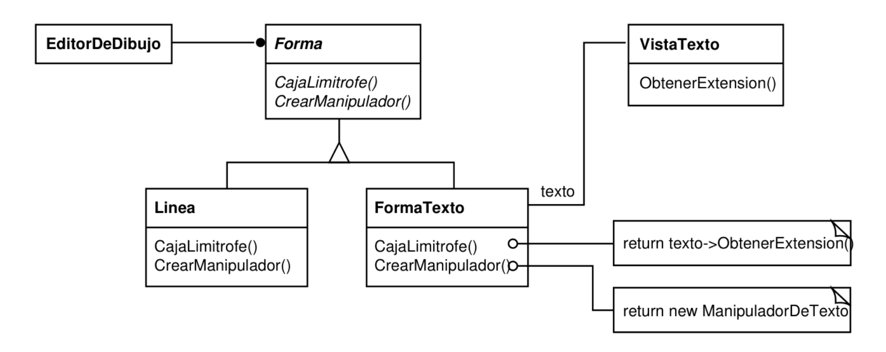
Las clases de formas geométricas elementales, como FormaLinea y FormaPoligono son bastante fáciles de implementar, ya que sus capacidades de dibujado y edición son intrínsecamente limitadas. Pero una subclase Texto que pueda mostrar y editar texto es considerablemente más difícil de implementar, ya que incluso la edición básica de texto implica actualizaciones de pantalla complicadas y gestión de búferes. A su vez, un toolkit comercial de interfaces de usuario podría proporcionar una clase VistaTexto

sofisticada para mostrar y editar texto. Lo que nos gustaría sería poder reutilizar VistaTexto para implementar FormaTexto, pero el toolkit no se diseñó con las clases Forma en mente. Por tanto, no podemos usar los objetos VistaTexto y Forma de manera intercambiable.

¿Cómo pueden funcionar clases existentes y no relacionadas, como VistaTexto, en una aplicación que espera clases con una interfaz diferente e incompatible? Podríamos cambiar la clase VistaTexto para que se ajustase a la interfaz Forma, pero eso no es una opción a menos que tengamos el código fuente del toolkit. Incluso aunque así fuese, no tendría sentido cambiar VistaTexto; el toolkit no debería tener que adoptar interfaces específicas del dominio sólo para que funcione una aplicación.

En vez de eso, podríamos definir FormaTexto para que *adapte* la interfaz VistaTexto a la de Forma. Podemos hacer esto de dos maneras: (1) heredando la interfaz de Forma y la implementación de VistaTexto, o (2) componiendo una instancia VistaTexto dentro de una FormaTexto e implementando FormaTexto en términos de la interfaz de VistaTexto. Ambos enfoques se corresponden con las versiones de clases y de objetos del patrón Adapter. Decimos entonces que FormaTexto es un **adaptador**.

El siguiente diagrama ilustra el caso del adaptador de objetos. Muestra cómo las peticiones a CajaLimitrofe, declarada en una clase Forma, se convierten en peticiones a ObtenerExtension, definida en VistaTexto. Puesto que FormaTexto adapta VistaTexto a la interfaz Forma, el editor de dibujo puede reutilizar la clase VistaTexto, que de otro modo sería incompatible.



Muchas veces el adaptador es responsable de la funcionalidad que

la clase adaptada no proporciona. El diagrama muestra cómo un adaptador puede llevar a cabo tales responsabilidades. El usuario debería ser capaz de "arrastrar" interactivamente cada objeto Forma a una nueva posición, pero VistaTexto no está diseñada para ello. FormaTexto puede añadir esta funcionalidad ausente implementando la operación de Forma CrearManipulador, que devuelve una instancia de la subclase Manipulador apropiada.

Manipulador es una clase abstracta para objetos que saben cómo hacer que una Forma responda a la entrada del usuario, como arrastrar la forma a una nueva posición. Hay subclases de Manipulador para diferentes formas: ManipuladorDeTexto, es la subclase correspondiente a FormaTexto. Devolviendo una instancia de ManipuladorDeTexto, FormaTexto añade la funcionalidad que VistaTexto no tiene y que es requerida por Forma.

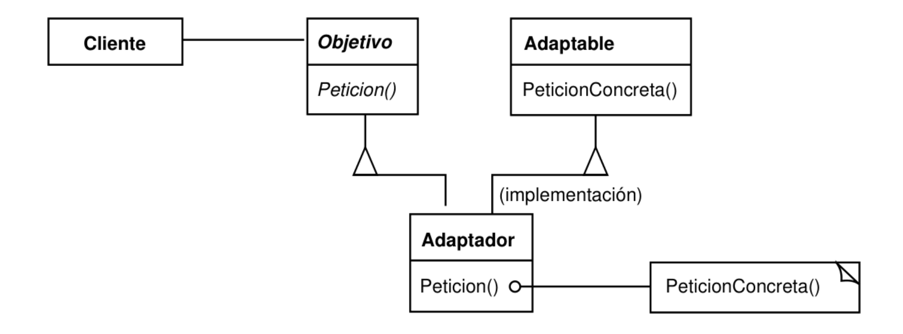
APLICABILIDAD

Debería usarse el patrón Adapter cuando

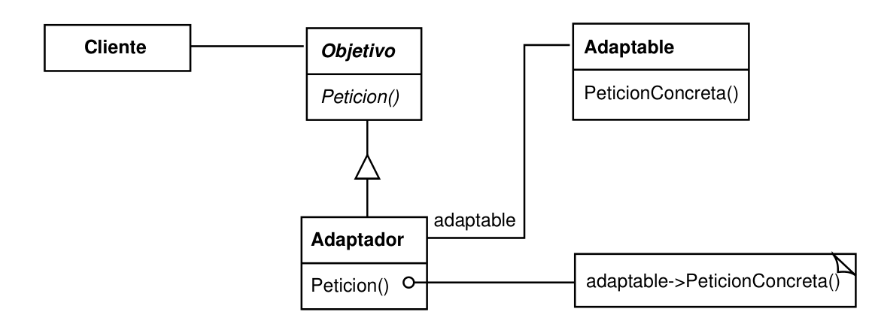
- se quiere usar una clase existente y su interfaz no concuerda con la que necesita.
- se quiere crear una clase reutilizable que coopere con clases no relacionadas o que no han sido previstas, es decir, clases que no tienen por qué tener interfaces compatibles.
- (*solamente en el caso de un adaptador de objetos*) es necesario usar varias subclases existentes, pero no resulta práctico adaptar su interfaz heredando de cada una de ellas. Un adaptador de objetos puede adaptar la interfaz de su clase padre.

ESTRUCTURA

Un adaptador de clases usa la herencia múltiple para adaptar una interfaz a otra:



Un adaptador de objetos se basa en la composición de objetos:



PARTICIPANTES

- **Objetivo** (Forma)
  - define la interfaz específica del dominio que usa el Cliente.
- **Cliente** (EditorDeDibujo)
  - colabora con objetos que se ajustan a la interfaz Objetivo.
- **Adaptable** (VistaTexto)
  - define una interfaz existente que necesita ser adaptada.
- **Adaptador** (FormaTexto)
  - adapta la interfaz de Adaptable a la interfaz Objetivo.

COLABORACIONES

Los clientes llaman a operaciones de una instancia de Adaptador. A

su vez, el adaptador llama a operaciones de Adaptable, que son las que satisfacen la petición.

CONSECUENCIAS

Los adaptadores de clases y de objetos tienen diferentes ventajas e inconvenientes. Un adaptador de clases

- adapta una clase Adaptable a Objetivo, pero se refiere únicamente a una clase Adaptable concreta. Por tanto, un adaptador de clases no nos servirá cuando lo que queremos es adaptar una clase y todas sus subclases.
- permite que Adaptador redefina parte del comportamiento de Adaptable, por ser Adaptador una subclase de Adaptable.
- introduce un solo objeto, y no se necesita ningún puntero de indirección adicional para obtener el objeto adaptado.

Por su parte, un adaptador de objetos

- permite que un mismo Adaptador funcione con muchos Adaptables —es decir, con el Adaptable en sí y todas sus subclases, en caso de que las tenga—. El Adaptador también puede añadir funcionalidad a todos los Adaptables a la vez.
- hace que sea más difícil redefinir el comportamiento de Adaptable. Se necesitará crear una subclase de Adaptable y hacer que el Adaptador se refiera a la subclase en vez de a la clase Adaptable en sí.

Éstas son otras cuestiones a tener en cuenta al usar el patrón Adaptador:

1. ¿*Cuánta adaptación hace el Adaptador*? Los adaptadores difieren en la cantidad de trabajo necesaria para adaptar Adaptable a la interfaz Objetivo. Hay una amplia gama de tareas posibles, que van desde la simple conversión de interfaces —por ejemplo, cambiar los nombres de las operaciones— a permitir un conjunto completamente nuevo de operaciones. La cantidad de trabajo que hace el Adaptador depende de lo parecida que sea la interfaz de Objetivo a la de Adaptable.

2. *Adaptadores conectables*. Una clase es más reutilizable cuando minimizamos las asunciones que deben hacer otras clases para usarla. Al hacer la adaptación de interfaces en una clase estamos eliminando la asunción de que otras clases ven la misma interfaz. Dicho de otro modo, la adaptación de interfaces nos permite incorporar nuestra clase a sistemas existentes que podrían esperar interfaces diferentes a la de la clase. ObjectWorks\Smalltalk [Par90] usa el término **adaptador conectable** para describir clases con adaptación de interfaces incorporada Pensemos en un útil[34] VisualizadorDeArboles que puede mostrar estructuras arbóreas gráficamente. Si éste fuera un útil de propósito especial para usar en una única aplicación podríamos obligar a los objetos que representa a que tuviesen una interfaz concreta: es decir, que todos descendiesen de una clase abstracta Arbol. Pero si quisiéramos hacer a VisualizadorDeArboles más reutilizable (supongamos que quisiéramos hacerlo parte de un toolkit de útiles), entonces ese requisito sería poco razonable. Las aplicaciones definirán sus propias clases para estructuras de árbol. No deberían estar obligadas a utilizar nuestra clase abstracta Arbol. Estructuras arbóreas diferentes tendrán diferentes interfaces.

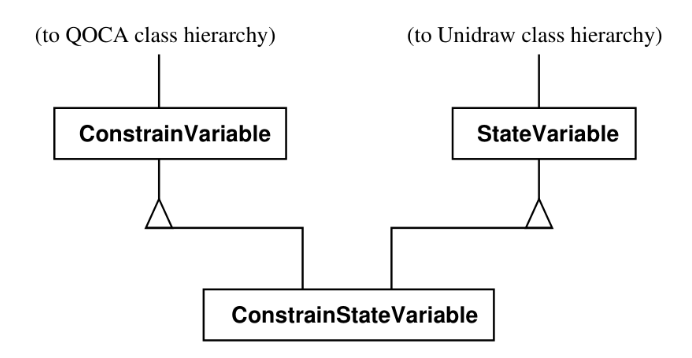
   Por ejemplo, en una jerarquía de directorios podría accederse a los hijos con una operación ObtenerSubdirectorios, mientras que en una jerarquía de clases la operación correspondiente podría llamarse ObtenerSubclases. Un útil VisualizadorDeArboles reutilizable debe ser capaz de representar ambos tipos de jerarquías, incluso aunque usen diferentes interfaces. En otras palabras, el VisualizadorDeArboles debería tener la adaptación de interfaces incorporada.

   En la sección de Implementación veremos diferentes modos de incorporar la adaptación de interfaces a las clases.

3. *Usar adaptadores bidireccionales para proporcionar*

transparencia. Un problema potencial de los adaptadores es que no son transparentes a todos los clientes. Un objeto adaptado ya no se ajusta a la interfaz Adaptable, por lo que no puede usarse tal cual en donde pudiera ir un objeto Adaptable. Los **adaptadores bidireccionales** pueden proporcionar esa transparencia. En concreto, resultan útiles cuando dos clientes distintos necesitan ver un objeto de distinta forma. Pensemos en el adaptador bidireccional que incorpora Unidraw, un framework de editores gráficos [VL90], y QOCA, un toolkit de resolución de problemas [HHMV92]. Ambos sistemas tienen clases que representan variables explícitamente: Unidraw tiene State Variable, y QOCA tiene ConstraintVariable. Para hacer que Unidraw funcione con QOCA, hay que adaptar ConstraintVariable a State Variable; para que QOCA propague soluciones a Unidraw, hay que adaptar State Variable a ConstraintVariable.

La solución consiste en un adaptador de clases bidireccional ConstraintStateVariable, una subclase de State Variable y ConstraintVariable, que adapta cada una de las dos interfaces a la otra. La herencia múltiple es una solución viable en este caso porque las interfaces de las clases adaptadas son sustancialmente diferentes. El adaptador de clases bidireccional se ajusta a las dos clases adaptadas y puede trabajar en cualquiera de los dos sistemas.



IMPLEMENTACIÓN

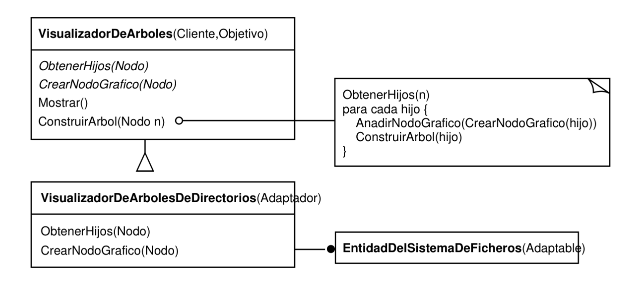
Aunque la implementación del patrón Adapter suele ser sencilla, éstas son algunas cuestiones que hay que tener en cuenta:

1. *Implementación de adaptadores de clases* en C++. En una implementación C++ de un adaptador de clases. Adaptador debería heredar públicamente de Objetivo y privadamente de Adaptable. Así, Adaptador sería un subtipo de Objetivo, pero no de Adaptable.
2. *Adaptadores conectables*. Veamos tres formas de implementar adaptadores conectables para el útil VisualizadorDeArboles descrito anteriormente, el cual puede mostrar una estructura jerárquica automáticamente.

El primer paso, que es común a las tres implementaciones estudiadas aquí, consiste en encontrar una interfaz “reducida” para Adaptable, es decir, el subconjunto más pequeño de operaciones que nos permita hacer la adaptación. Una interfaz reducida consistente en sólo un par de operaciones es más fácil de adaptar que una interfaz con docenas de operaciones. Para VisualizadorDeArboles, el adaptable es cualquier estructura jerárquica. Una interfaz minimalista podría incluir dos operaciones, una que defina cómo presentar un nodo gráficamente en la estructura jerárquica, y otra que recupere los hijos del nodo.

La interfaz reducida nos lleva a tres enfoques de implementación:

a) *Usar operaciones abstractas*. Definir las correspondientes operaciones abstractas para la interfaz reducida de Adaptable en la clase VisualizadorDeArboles. Las subclases deben implementar las operaciones abstractas y adaptar el objeto estructurado jerárquicamente. Por ejemplo, una subclase VisualizadorDeArbolDeDirectorios implementará estas operaciones accediendo a la estructura de directorios.

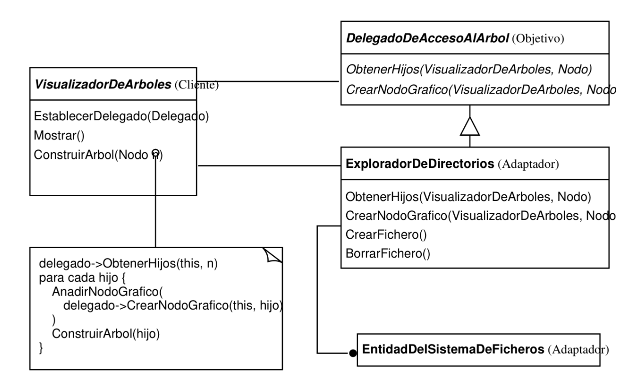


VisualizadorDeArbolDeDirectorios especializa la interfaz reducida de manera que pueda representar y mostrar estructuras de directorio formadas por objetos EntidadDelSistemaDeFicheros.

b) *Usar objetos delegados*. Con este enfoque, VisualizadorDeArboles reenvía las peticiones para acceder a la estructura jerárquica a un objeto **delegado**. VisualizadorDeArboles puede usar una estrategia de adaptación diferente cambiando el objeto delegado.

Por ejemplo, supongamos que existe un ExploradorDeDirectorios que usa un VisualizadorDeArboles. ExploradorDeDirectorios podría ser un buen delegado para adaptar Visualizador De Arboles a la estructura jerárquica de directorios. En lenguajes dinámicamente tipados, como Smalltalk u Objetive C, este enfoque sólo necesita una interfaz para registrar el delegado con el adaptador. A continuación, VisualizadorDeArboles simplemente reenvía las peticiones al delegado. NEXTSTEP [Add94] usa este enfoque intensivamente para reducir el uso de subclases. Los lenguajes estáticamente tipados, como C++, necesitan una definición de interfaz explícita para el delegado. Podemos especificar dicha interfaz poniendo la interfaz reducida que necesita VisualizadorDeArboles en una clase abstracta DelegadoDeAccesoAlArbol. A continuación podemos combinar esta interfaz con el delegado de nuestra elección —ExploradorDeDirectorios, en este caso— a través de la herencia. Usaremos herencia simple si el ExploradorDeDirectorios no tiene ninguna clase padre, y

herencia múltiple en caso de que la tenga. Combinar clases de esta forma es más fácil que introducir una nueva subclase VisualizadorDeDirectorios e implementar sus operaciones individualmente.



c) *Adaptadores parametrizados*. La forma más común de permitir adaptadores conectables en Smalltalk es parametrizar un adaptador con uno o más bloques. La construcción *block* permite la adaptación sin necesidad de subclases. Un bloque puede adaptar una petición y el adaptador puede almacenar un bloque para cada petición individual. En nuestro ejemplo, esto quiere decir que VisualizadorDeArboles guarda un bloque para convertir un nodo en un NodoGrafico y otro bloque para acceder a los hijos de un nodo.

Por ejemplo, para crear VisualizadorDeArboles en una jerarquía de directorios, escribimos

```
VisualizadorDeDirectorios :=
     (VisualizadorDeArboles       on:
raiz)
          obtenerBloqueHijos:
               [:nodo      ¦      nodo
obtenerSubdirectorios]
          crearBloqueNodoGrafico:
               [:nodo      ¦      nodo
crearNodoGrafico].
```

Si estamos adaptando una interfaz a una clase, este enfoque ofrece una buena alternativa a la herencia.

CÓDIGO DE EJEMPLO

Daremos un breve esbozo de la implementación de adaptadores de clases y de objetos para el ejemplo de la sección de Motivación, comenzando con las clases Forma y VistaTexto.

```
class Forma {
public:
     Forma();
     virtual void CajaLimitrofe(
          Punto&  inferiorIzquierdo,  Puntos
superiorDerecho
     ) const;
     virtual  Manipulador*  CrearManipulador()
const;
};

class VistaTexto {
public:
     VistaTexto();
     void  ObtenerOrigen(Coord&  x,  Coord&  y)
const;
     void  ObtenerArea(Coord&  ancho,  Coord&
alto) const;
     virtual bool EstaVacia() const;
};
```

Forma determina sus límites mediante una caja definida por sus esquinas opuestas. VistaTexto, por el contrario, está definida por un origen, un alto y un ancho. Forma también define una operación CrearManipulador para crear un Manipulador[35], el cual sabe cómo mover una forma cuando ésta es manipulada por el usuario. VistaTexto no tiene una operación equivalente. La clase FormaTexto es un adaptador entre estas interfaces diferentes.

Un adaptador de clases usa la herencia múltiple para adaptar interfaces. La clave de los adaptadores de clases es usar una rama de la herencia para heredar la interfaz y otra para heredar la implementación. La forma normal de hacer esta distinción en C++ es heredar públicamente la interfaz y privadamente la

implementación. Usaremos este convenio para definir el adaptador FormaTexto.

```
class FormaTexto : public Forma, private
VistaTexto {
public:
     FormaTexto();

     virtual void CajaLimitrofe(
          Punto& inferiorIzquierdo, Punto&
superiorDerecho
     ) const;
     virtual bool EstaVacia() const;
     virtual Manipulador* CrearManipulador()
const;
};
```

La operación CajaLimitrofe convierte la interfaz de VistaTexto para que se ajuste a la de Forma.

```
void FormaTexto::CajaLimitrofe (
     Punto&    inferiorIzquierdo,    Punto&
superiorDerecho
) const {
     Coord inferior, izquierda, ancho, alto;

     ObtenerOrigen(inferior, izquierda);
     ObtenerArea(ancho, alto);

     inferiorIzquierdo   =   Punto(inferior,
izquierda);
     superiorDerecho = Punto(inferior + alto,
izquierda + ancho);
}
```

La operación EstaVacia ilustra el reenvío directo de peticiones que es común en las implementaciones de adaptadores:

```
bool FormaTexto::EstaVacia () const {
     return VistaTexto::EstaVacia();
}
```

Por último, definiremos CrearManipulador (que no es admitida por VistaTexto) desde cero. Suponemos que ya hemos implementado una clase ManipuladorDeTexto que permite la manipulación de una FormaTexto.

```
Manipulador* FormaTexto::CrearManipulador ()
const {
     return new ManipuladorDeTexto(this);
}
```

El objeto adaptador usa composición de objetos para combinar clases con diferentes interfaces. Con este enfoque, el adaptador FormaTexto mantiene un puntero a VistaTexto.

```
class FormaTexto : public Forma {
public:
     FormaTexto(VistaTexto*);

     virtual void CajaLimitrofe(
          Punto&  inferiorIzquierdo,  Punto&
superiorDerecho
     ) const;
     virtual bool EstaVacia() const;
     virtual Manipulador* CrearManipulador()
const;
private:
     VistaTexto* _texto;
};
```

FormaTexto debe inicializar el puntero a la instancia de VistaTexto, y lo hace en el constructor. También tiene que llamar a las operaciones de su objeto VistaTexto cada vez que se llama a las suyas propias. En este ejemplo suponemos que el cliente crea el objeto VistaTexto y lo pasa al constructor de FormaTexto:

```
FormaTexto::FormaTexto (VistaTexto* t) {
     _texto = t;
}

void FormaTexto::CajaLimitrofe (
```

```
     Punto&     inferiorIzquierdo,     Puntos
superiorDerecho
) const {
     Coord inferior, izquierda, ancho, alto;

     _texto->ObtenerOrigen(inferior,
izquierda);
     _texto->ObtenerArea(ancho, alto);
     inferiorIzquierdo    =    Punto(inferior,
izquierda);
     superiorDerecho = Punto(inferior + alto,
izquierda + ancho);
}

bool FormaTexto::EstaVacia () const {
     return _texto->EstaVacia();
}
```

La implementación de CrearManipulador no cambia respecto de la versión del adaptador de clases, ya que está implementado desde cero y no reutiliza nada de la funcionalidad existente de VistaTexto.

```
Manipulador*  FormaTexto::CrearManipulador  ()
const {
     return new ManipuladorDeTexto(this);
}
```

Compárese este código con el caso del adaptador de clases. El adaptador de objetos requiere un poco más de esfuerzo a la hora de escribirlo, pero también es más flexible. Por ejemplo, la versión del adaptador de objetos de FormaTexto funcionará igualmente bien con subclases de VistaTexto —el cliente simplemente pasa una instancia de una subclase de VistaTexto al constructor de FormaTexto—.

USOS CONOCIDOS

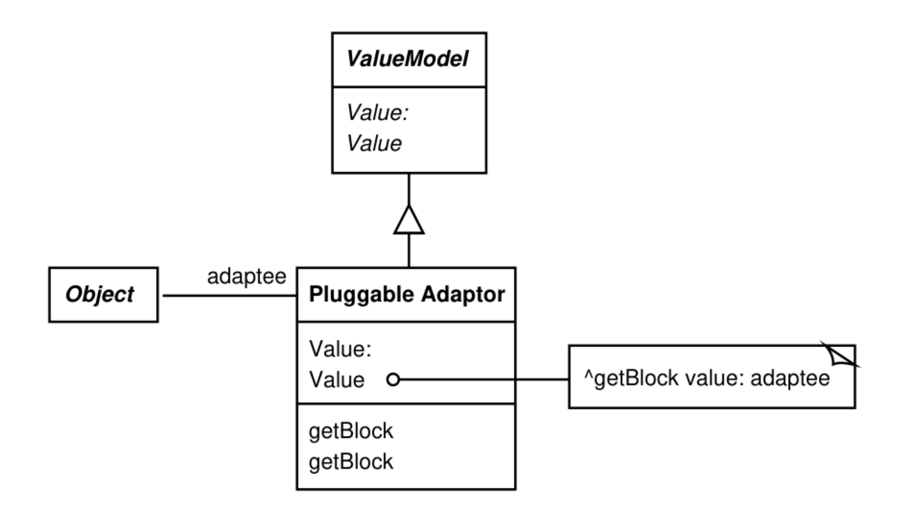
El ejemplo de la sección Motivación viene de ET++Draw, una aplicación de dibujo basada en ET++ [WGM88]. ET++Draw reutiliza las clases de ET++ para editar texto mediante una clase adaptadora TextShape.

Interviews 2.6 define una clase abstracta Interactor para elementos de la interfaz de usuario como barras de desplazamiento, botones y menús [VL88]. También define una clase abstracta Graphic para objetos gráficos estructurados, tales como líneas, círculos, polígonos y *splines*. Tanto Interactor como Graphic tienen representación visual, pero ambos tienen diferentes interfaces e implementaciones (no comparten una clase padre común) y son por tanto incompatibles —no podemos insertar directamente un objeto gráfico estructurado en, pongamos por caso, un cuadro de diálogo —.

En vez de eso, Interviews 2.6 define un adaptador de objetos llamado GraphicBlock, una subclase de Interactor que contiene una instancia de Graphic. El GraphicBlock adapta la interfaz de la clase Graphic a la de Interactor. El GraphicBlock permite que se pueda mostrar, desplazar y hacer *zoom* sobre una instancia de Graphic dentro de un estructura Interactor.

Los adaptadores conectables son frecuentes en ObjectWorks \Smalltalk [Par90]. El Smalltalk estándar define una clase ValueModel para las vistas que muestran un valor único. ValueModel define una interfaz value, value: para acceder al valor. Éstos son métodos abstractos. Los implementadores de la aplicación acceden al valor con nombres más específicos del dominio, como ancho y ancho:, pero no deberían tener que crear una subclase de ValueModel para adaptar dichos nombres específicos del dominio a la interfaz ValueModel.

En lugar de eso, ObjectWorks\Smalltalk incluye una subclase de ValueModel llamada PluggableAdaptor. Un objeto Pluggable Adaptor adapta otros objetos a la interfaz de ValueModel (value, va lúe:). Puede ser parametrizado con bloques para obtener y establecer el valor deseado. PluggableAdaptor utiliza internamente estos bloques para implementar la interfaz value, value:. PluggableAdaptor también permite pasar nombres directamente en el selector (por ejemplo, ancho, ancho:) por conveniencia sintáctica, convirtiendo estos selectores en los bloques correspondientes de forma automáticamente.



Otro ejemplo de ObjectWorks\Smalltalk es la clase Table Adaptor. Un Table Adaptor puede adaptar una secuencia de objetos a una presentación tabular. La tabla muestra un objeto por fila. El cliente parametriza TableAdaptor con el conjunto de mensajes que puede usar una tabla para obtener de un objeto los valores de las columnas.

Algunas clases del AppKit de NeXT [Add94] usan delegación de objetos para llevar a cabo adaptación de interfaces. Un ejemplo es la clase NXBrowser que puede mostrar listas jerárquicas de datos. NXBrowser usa un objeto delegado para acceder a los datos y adaptarlos.

El "Matrimonio de Conveniencia" de Meyer [Mey88] es una forma de adaptador de clases. Meyer describe cómo una clase PilaFija adapta la implementación de una clase Array a la interfaz de una clase Pila. El resultado es una pila que contiene un número fijo de entradas.

PATRONES RELACIONADOS

El patrón Bridge (141) tiene una estructura similar a un adaptador de objetos, pero con un propósito diferente: está pensado para separar una interfaz de su implementación, de manera que ambos puedan cambiar fácilmente y de forma independiente uno del otro, mientras que un adaptador está pensado para cambiar la interfaz de un objeto existente.

El patrón Decorator (161) decora otro objeto sin cambiar su interfaz. Un decorador es por tanto más transparente a la aplicación que un adaptador. Como resultado, el patrón Decorator permite la composición recursiva, lo que no es posible con adaptadores puros.

El patrón Proxy (191) define un representante o sustituto de otro objeto sin cambiar su interfaz.

Estructural de Objetos

BRIDGE (Puente)

PROPÓSITO

Desacopla una abstracción de su implementación, de modo que ambas puedan variar de forma independiente.

TAMBIÉN CONOCIDO COMO

Handle/Body (Manejador/Cuerpo)

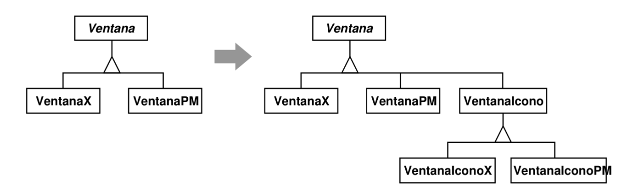
MOTIVACIÓN

Cuando una abstracción puede tener varias implementaciones posibles, la forma más habitual de darles cabida es mediante la herencia. Una clase abstracta define la interfaz de la abstracción, y las subclases concretas la implementan de distintas formas. Pero este enfoque no siempre es lo bastante flexible. La herencia liga una implementación a la abstracción de forma permanente, lo que dificulta modificar, extender y reutilizar abstracciones e implementaciones de forma independiente.

Pensemos en la implementación de una abstracción portable Ventana en un toolkit de interfaces de usuario. Esta abstracción debería permitirnos escribir aplicaciones que funcionen, por ejemplo, tanto en el Sistema de Ventanas X como en Presentation Manager de IBM (PM). Mediante la herencia podríamos definir una clase abstracta Ventana y subclases VentanaX y VentanaPM que implementen la interfaz Ventana para las distintas plataformas. Pero este enfoque tiene dos inconvenientes:

1. No es conveniente extender la abstracción Ventana para cubrir diferentes tipos de ventanas o nuevas plataformas. Imaginemos una subclase VentanaIcono, que especializa la abstracción Ventana para representar iconos. Para admitir

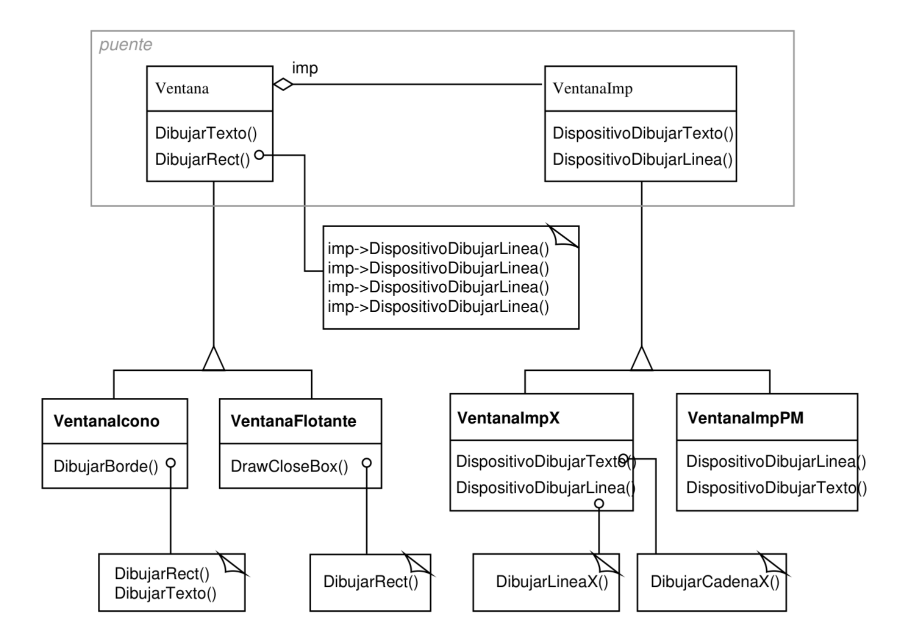
este tipo de ventanas en ambas plataformas debemos implementar *dos* nuevas clases, VentanaIconoX y VentanaIconoPM. Y lo que es peor, tendremos que definir dos clases para *cada* tipo de ventana. Dar cabida a una tercera plataforma requeriría otra nueva subclase de Ventana para cada tipo de ventana.



2. Hace que el código sea dependiente de la plataforma. Cada vez que un cliente crea una ventana, se crea una clase concreta que tiene una determinada implementación. Por ejemplo, crear un objeto VentanaX liga la abstracción Ventana a la implementación para X Window, lo que vuelve al código del cliente dependiente de dicha implementación. A su vez, esto hace que sea más difícil portar el código cliente a otras plataformas.

   Los clientes deberían ser capaces de crear una ventana sin someterse a una implementación concreta. Lo único que tendría que depender de la plataforma en la que se ejecuta la aplicación es la implementación de la ventana. Por tanto, el código cliente debería crear ventanas sin hacer mención a plataformas concretas.

El patrón Bridge resuelve estos problemas situando la abstracción Ventana y su implementación en jerarquías de clases separadas. Hay una jerarquía de clases para las interfaces de las ventanas (Ventana. VentanaIcono, VentanaFlotante) y otra jerarquía aparte para las implementaciones específicas de cada plataforma, teniendo a VentanaImp **[36]** como su raíz. Por ejemplo, la subclase VentanaImpX proporciona una implementación basada en el sistema de ventanas X Window.



Todas las operaciones de las subclases de Ventana se implementan en términos de operaciones abstractas de la interfaz VentanaImp. Esto desacopla las abstracciones ventana de las diferentes implementaciones específicas de cada plataforma. Nos referiremos a la relación entre Ventana y VentanaImp como un **puente (bridge)**, porque une a la abstracción con su implementación, permitiendo que ambas varíen de forma independiente.

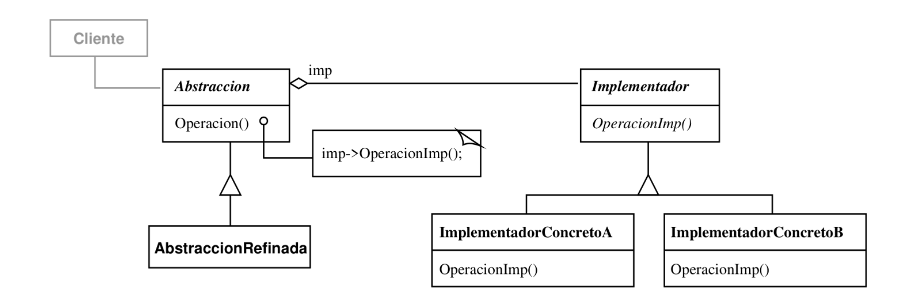
APLICABILIDAD

Use el patrón Bridge cuando

- quiera evitar un enlace permanente entre una abstracción y su implementación. Por ejemplo, cuando debe seleccionarse o cambiarse la implementación en tiempo de ejecución.
- tanto las abstracciones como sus implementaciones deberían ser extensibles mediante subclases. En este caso, el patrón Bridge permite combinar las diferentes abstracciones y sus implementaciones, y extenderlas independientemente.
- los cambios en la implementación de una abstracción no deberían tener impacto en los clientes: es decir, su código no tendría que ser recompilado.

- (C++) quiera ocultar completamente a los clientes la implementación de una abstracción. En C++ la representación de una clase es visible en la interfaz de la misma.
- tenga una proliferación de clases como la mostrada en el primer diagrama de la sección Motivación. Una jerarquía de clases tal pone de manifiesto la necesidad de dividir un objeto en dos partes. Rumbaugh usa el término “generalizaciones anidadas” [RBP+91j para referirse a dichas jerarquías de clases.
- quiera compartir una implementación entre varios objetos (tal vez usando un contador de referencias) y este hecho deba permanecer oculto al cliente. Un ejemplo sencillo es la clase String de Coplien [Cop92], donde varios objetos pueden compartir la misma representación de una cadena (StringRep).

ESTRUCTURA



PARTICIPANTES

- **Abstracción** (Ventana)
  - define la interfaz de la abstracción.
  - mantiene una referencia a un objeto de tipo Implementador.
- **AbstraccionRefinada** (VentanaIcono)
  - extiende la interfaz definida por Abstracción.
- **Implementador** (VentanaImp)

- define la interfaz de las clases de implementación. Esta interfaz no tiene por qué corresponderse exactamente con la de Abstracción; de hecho, ambas interfaces pueden ser muy distintas. Normalmente la interfaz Implementador sólo proporciona operaciones primitivas, y Abstracción define operaciones de más alto nivel basadas en dichas primitivas.

- **ImplementadorConcreto** (VentanaImpX, VentanaImpPM)
  - implementa la interfaz Implementador y define su implementación concreta.

COLABORACIONES

Abstracción redirige las peticiones del cliente a su objeto Implementador.

CONSECUENCIAS

El patrón Bridge tiene las siguientes consecuencias:

1. *Desacopla la interfaz y la implementación*. No une permanentemente una implementación a una interfaz, sino que la implementación puede configurarse en tiempo de ejecución. Incluso es posible que un objeto cambie su implementación en tiempo de ejecución.

   Desacoplar Abstracción e Implementador también elimina de la implementación dependencias de tiempo de compilación. Ahora, cambiar una clase ya no requiere recompilar la clase Abstracción y sus clientes. Esta propiedad es esencial cuando debemos asegurar la compatibilidad binaria entre distintas versiones de una biblioteca de clases.

   Además, este desacoplamiento potencia una división en capas que puede dar lugar a sistemas mejor estructurados. La parte de alto nivel de un sistema sólo tiene que conocer a Abstracción y a Implementador.

2. *Mejora la extensibilidad*. Podemos extender las jerarquías

de Abstracción y de Implementador de forma independiente.

3. *Oculta detalles de implementación a los clientes*. Podemos aislar a los clientes de los detalles de implementación, como el compartimiento de objetos implementadores y el correspondiente mecanismo de conteo de referencias (si es que hay alguno).

IMPLEMENTACIÓN

Al aplicar el patrón Bridge hemos de tener en cuenta las siguientes cuestiones de implementación:

1. *Un único implementador*. En situaciones en las que sólo hay una implementación, no es necesario crear una clase abstracta Implementador. Éste es un caso degenerado del patrón Bridge, cuando hay una relación uno-a-uno entre Abstracción e Implementador. Sin embargo, esta separación sigue siendo útil cuando un cambio en la implementación de una clase no debe afectar a sus clientes existentes, es decir, que éstos no deberían tener que ser recompilados, sino sólo vueltos a enlazar.

   Carolan [Car89] usa la expresión "Gato de Cheshire" para describir dicha separación. En C++ se puede definir la interfaz de la clase Implementador en un fichero de cabecera privado que no se proporciona a los clientes. Esto permite ocultar por completo la implementación de la clase a los clientes.

2. *Crear el objeto Implementador apropiado*. ¿Cómo, cuándo y dónde se decide de qué clase Implementador se van a crear las instancias cuando hay más de una? Si Abstracción conoce a todas las clases ImplementadorConcreto, puede crear una instancia de una de ellas en su constructor; puede decidir de cuál basándose en los parámetros pasados a su constructor. Por ejemplo, si la clase de una colección admite varias implementaciones, la decisión puede estar basada en el tamaño de la colección. Se puede

usar una lista enlazada para colecciones pequeñas y una tabla de dispersión (*hash*) para colecciones grandes.

Otro enfoque consiste en elegir inicialmente una implementación predeterminada y cambiarla después en función de su uso. Si, por ejemplo, la colección crece más allá de un cierto límite, puede cambiar su implementación por otra que resulte más apropiada para un gran número de elementos.

También es posible delegar totalmente la decisión en otro objeto. En el ejemplo de la Ventana/VentanaImp se puede introducir un objeto fábrica (*véase* el patrón Abstract Factory (79)) cuya única misión sea encapsular detalles de implementación. La fábrica sabe qué tipo de objeto VentanaImp crear para la plataforma en uso; una Ventana simplemente solicita una VentanaImp, y devuelve el tipo adecuado de ésta. Una ventaja de este enfoque es que Abstracción no está acoplada directamente a ninguna de las clases Implementador.

3. *Compartimiento de Implementadores*. Coplien ilustra cómo se puede usar el modismo de C++ Handle/Body (Manejador/Cuerpo) para compartir implementaciones entre varios objetos [Cop92]. El Cuerpo tiene un contador de referencias que es incrementado y disminuido por la clase Manejador. El código para asignar manejadores con cuerpos compartidos tiene la siguiente forma general:

```
Manejador&       Manejador::operator»
(const Manejador& otro) {
     otro._cuerpo->Ref();
     _cuerpo->QuitarRef();

     if                      (_cuerpo-
>ContadorReferencias() == 0) {
          delete _cuerpo;
     }
     _cuerpo = otro._cuerpo;

     return *this;
```

```
        }
```

4. *Uso de la herencia múltiple*. Se puede utilizar herencia múltiple en C++ para combinar una interfaz con su implementación [Mar91j. Por ejemplo, una clase puede heredar públicamente de Abstracción y privadamente de ImplementadorConcreto. Pero dado que este enfoque se basa en la herencia estática, está asociando permanentemente una implementación a su interfaz. Por tanto, no se puede implementar un verdadero Bridge usando herencia estática, al menos no en C++.

CÓDIGO DE EJEMPLO

El siguiente código C++ implementa el ejemplo Ventana/ VentanaImp de la sección de Motivación. La clase Ventana define la abstracción de ventana para las aplicaciones clientes:

```
class Ventana {
public:
     Ventana(Vista* contenido);

     // peticiones manejadas por la ventana
     virtual void DibujarContenido();
     virtual void Abrir();
     virtual void Cerrar();
     virtual void Minimizar();
     virtual void Maximizar();

     //    peticiones    reenviadas    a    su
implementación
     virtual    void    EstablecerOrigen(const
Punto& en);
     virtual  void  EstablecerArea(constPunto&
area);
     virtual void TraerAlFrente();
     virtual void EnviarAlFondo();

     virtual  void  DibujarLinea(const  Punto&,
const Punto&);
     virtual  void  DibujarRect(const  Punto&,
const Punto&);
     virtual    void    DibujarPoligono(const
```

```
Punto[], int n);
     virtual  void  DibujarTexto(const  char*,
const Punto&);

protected:
     VentanaImp* ObtenerVentanaImp();
     Vista* ObtenerVista();

private:
     VentanaImp* _imp;
     Vista* _contenido; // el contenido de la
ventana
};
```

Ventana mantiene una referencia a VentanaImp, la clase abstracta que declara una interfaz para el sistema de ventanas subyacente.

```
class VentanaImp {
public:
     virtual void ImpSuperior() = 0;
     virtual void ImpInferior() = 0;
     virtual  void  ImpEstablecerArea  (const
Punto&) = 0;
     virtual  void  ImpEstablecerOrigen(const
Punto&) = 0;

     virtual   void   DispositivoRect(Coord,
Coord, Coord, Coord) = 0;
     virtual   void   DispositivoTexto(const
char*, Coord, Coord) = 0;
     virtual void DispositivoMapaDeBits(const
char*, Coord, Coord) = 0;
     // muchas más funciones para dibujar en
las ventanas…
protected:
     VentanaImp();
};
```

Las subclases de Ventana definen los diferentes tipos de ventanas que puede usar la aplicación, como ventanas de aplicación, iconos, ventanas flotantes para los diálogos, paletas de herramientas flotantes, etcétera.

Por ejemplo, VentanaAplicacion implementará DibujarContenido

para que dibuje su instancia de Vista:

```
class VentanaAplicacion : public Ventana {
public:
     // …
     virtual void DibujarContenido() {
};

void VentanaAplicacion::DibujarContenido () {
     ObtenerVista()->DlbujarEn(this);
}
```

VentanaIcono almacena el nombre de un mapa de bits con el icono que muestra…

```
class VentanaIcono : public Ventana {
public:
     // …
     virtual void DibujarContenido();
private:
     const char* _nombreMapaDaBits;
};
```

… e implementa DibujarContenido para que dibuje el mapa de bits en la ventana:

```
void VentanaIcono::DibujarContenido() {
     VentanaImp* imp = ObtenerVentanaImp();
     if (imp != 8) {
          imp-
>DispositivoMapaDeBits(_nombreMapaDeBits,
0.0, 0.0);
     }
}
```

Hay muchas otras posibles variantes de Ventana. Una VentanaFlotante puede necesitar comunicarse con la ventana principal que la creó; de ah(que contenga una referencia a dicha ventana. Una VentanaPaleta siempre flota sobre otras ventanas. Una

VentanaDeIconos contiene objetos VentanaIcono y los coloca como es debido.

Las operaciones de Ventana se definen en términos de la interfaz VentanaImp. Por ejemplo, DibujarRect extrae cuatro coordenadas a partir de sus dos parámetros Punto antes de llamar a la operación de VentanaImp que dibuja el rectángulo en la ventana:

```
void Ventana::DibujarRect (const Punto& p1,
const Punto& p2) {
     VentanaImp* imp = ObtenerVentanaImp();
     imp->DispositivoRect(p1.X(),     p1.Y(),
p2.X(), p2.Y());
}
```

Las subclases concretas de VentanaImp admiten diferentes sistemas de ventanas. La subclase VentanaImpX admite el sistema de ventanas X:

```
class VentanaImpX : public VentanaImp {
public:
     VentanaImpX();

     virtual   void   DispositivoRect(Coord,
Coord, Coord, Coord);
     // el resto de la interfaz pública…
private:
     //  estado  especifico  del  sistema  de
ventanas X, incluyendo:
     Display* _pantalla;
     Drawable _idVentana; // identificador de
ventana
     GC _cg;// contexto gráfico do la ventana
};
```

Para Presentation Manager (PM) definimos la clase VentanaImpPM:

```
class VentanaImpPM : public VentanaImp {
public:
     VentanaImpPM();
     virtual   void   DispositivoRect(Coord,
```

```
Coord, Coord, Coord);

     // el resto de la interfaz pública…
private:
     //  estado  especifico  del  sistema  de
ventanas PM, incluyendo:
     HPS _hps;
};
```

Estas subclases implementan las operaciones de VentanaImp en términos de las primitivas del sistema de ventanas. Por ejemplo, DispositivoRect se implemento para X como sigue:

```
void VentanaImpX::DispositivoRect (
     Coord x0, Coord y0, Coord x1, Coord y1
) {
     int x = round(min(x0, x1));
     int y = round(min(y0, y1));
     int ancho = round(abs(x0 - x1));
     int alto = round(abs(y0 - y1));
     XDrawRectangle(_pantalla,    _idVentana,
_cg, x, y, ancho, alto);
}
```

La implementación para PM podría parecerse a:

```
void VentanaImpPM::DispositivoRect (
     Coord x0, Coord y0, Coord x1, Coord y1
) {
     Coord izquierda = min(x0, x1);
     Coord derecha = max(x0, x1);
     Coord abajo = min(y0, y1);
     Coord arriba = max(y0, y1);

     PPOINTL punto[4];

     punto[0].x  =  izquierda;  punto[0].y  =
arriba;
     punto[1].x  =  derecha;  punto[1].y  =
arriba;
     punto[2].x  =  derecha;  punto[2].y  =
abajo;
     punto[3].x  =  izquierda;  punto[3].y  =
```

```
abajo;

     if (
          (GpiBeginPath(_hps,  1L)  ==  false)
¦¦
          (GpiSetCurrentPosition(_hps,
&punto[3]) == false) ¦¦
          (GpiPolyLine(_hps,  4L,  punto)  ==
GPI_ERR0R) ¦¦
          (GpiEndPath(_hps) == false)
     ) {
          // notificar error
     } else {
          GpiStrokePath(_hps, 1L, 0L);
     }
}
```

¿Cómo obtiene una ventana una instancia de la subclase correcta de VentanaImp? En este ejemplo, supondremos que es Ventana quien tiene esa responsabilidad. Su operación ObtenerVentanaImp obtiene la instancia correcta de una fábrica abstracta (*véase* el patrón Abstract Factory (79)) que encapsula todos los detalles del sistema de ventanas.

```
VentanaImp* Ventana::ObtenerVentanaImp () {
     if (_imp == 0) {
          _imp                              =
FabricaSistemaDeVentanas::Instancia()-
>HacerWindowImp();
     }
     return _imp;
}
```

FabricaSistemaDeVentanas:: Instance() devuelve una fábrica abstracta que produce todos los objetos específicos del sistema de ventanas. Por simplicidad hemos hecho que sea un Singleton (119) y hemos dejado que la clase Ventana acceda directamente a la fábrica.

USOS CONOCIDOS

El ejemplo de Ventana que acabamos de ver proviene de ET++ [WGM88]. En ET++ una VentanaImp se denomina “WindowPort” y tiene subclases tales como XWindowPort y SunWindowPort. El objeto Window crea su correspondiente objeto Implementador pidiéndoselo a una factoría abstracta llamada “WindowSystem”. WindowSystem proporciona una interfaz para crear objetos dependientes de la plataforma, como fuentes, cursores, mapas de bits, etcétera.

El diseño Window/WindowPort de ET++ extiende el patrón Bridge en el sentido de que WindowPort también mantiene una referencia a Window. La clase de implementación WindowPort usa dicha referencia para notificar a Window eventos específicos de WindowPort: la llegada de eventos de entrada, cambios del tamaño de pantalla, etc.

Both Coplien [Cop92] y Stroustrup [Str91] mencionan las clases Manejador (*Handler*) y dan algunos ejemplos de ellas. Sus ejemplos hacen hincapié en aspectos de la gestión de memoria, como compartir representaciones de cadenas y permitir objetos de tamaño variable. Nuestra atención se centra más en permitir que se puedan extender la abstracción y la implementación independientemente una de la otra.

libg++ [Lea88] define clases que implementan estructuras de datos comunes, tales como Set, LinkedSet, HashSet, LinkedList y HashTable. Set es una clase abstracta que define una abstracción de un conjunto, mientras que LinkedList y HashTable son implementadores concretos de una lista enlazada y una tabla de dispersión, respectivamente. LinkedSet y HashSet son implementadores de Set que unen a Set con sus equivalentes concretos LinkedList y HashTable. Éste es un ejemplo de un puente degenerado, porque no hay una clase abstracta Implementador.

El AppKit de NeXT [Add94] usa el patrón Bridge en la implementación y visualización de imágenes gráficas. Una imagen se puede representar de varias formas diferentes. La representación óptima de una imagen depende de las características de un dispositivo de visualización, concretamente de su capacidad de color y de su resolución. Sin la ayuda de AppKit los desarrolladores tendrían que determinar qué implementación usar bajo varias circunstancias en cada aplicación.

Para aliviar a los desarrolladores de esta responsabilidad, AppKit proporciona un puente para NXImage/NXImageRep. NXImage define la interfaz para manipular imágenes. La implementación de las imágenes se define en una jerarquía de clases separada NXImageRep que tiene subclases como NXEPSImageRep, NXCachedImageRep y NXBitMapImageRep. NXImage mantiene una referencia a uno o más objetos NXImageRep. Si hay más de una implementación de una imagen, NXImage selecciona la más apropiada para el dispositivo de visualización actual. NXImage es incluso capaz de convertir una implementación en otra si es necesario. El aspecto interesante de esta variante del Bridge es que NXImage puede almacenar más de una implementación de NXImageRep al mismo tiempo.

PATRONES RELACIONADOS

El patrón Abstract Factory (79) puede crear y configurar un Bridge.

El patrón Adapter (131) está orientado a conseguir que trabajen juntas clases que no están relacionadas. Normalmente se aplica a sistemas que ya han sido diseñados. El patrón Bridge, por otro lado, se usa al comenzar un diseño para permitir que abstracciones e implementaciones vanen independientemente unas de otras.

COMPOSITE (Compuesto)

PROPÓSITO

Compone objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.

MOTIVACIÓN

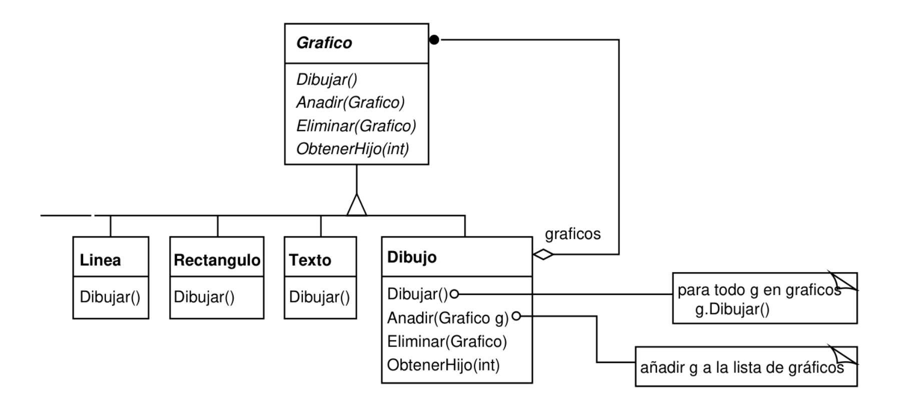
Las aplicaciones gráficas como los editores de dibujo y los sistemas de diseño de circuitos permiten a los usuarios construir diagramas complejos a partir de componentes simples. El usuario puede agrupar componentes para formar componentes más grandes, que a su vez pueden agruparse para formar componentes aún mayores. Una implementación simple podría definir clases para primitivas gráficas como Texto y Linea, más otras clases que actúen como contenedoras de estas primitivas.

Pero hay un problema con este enfoque: el código que usa estas clases debe tratar de forma diferente a los objetos primitivos y a los contenedores, incluso aunque la mayor parte del tiempo el usuario los trate de forma idéntica. Tener que distinguir entre estos objetos hace que la aplicación sea más compleja. El patrón Composite describe cómo usar la composición recursiva para que los clientes no tengan que hacer esta distinción.

La clave del patrón Composite es una clase abstracta que representa *tanto* a primitivas *como* a sus contenedores. Para el sistema gráfico, esta clase es Gráfico. Gráfico declara operaciones como Dibujar que son específicas de objetos gráficos. También declara operaciones que comparten todos los objetos compuestos, tales como operaciones para acceder a sus hijos y para gestionarlos.

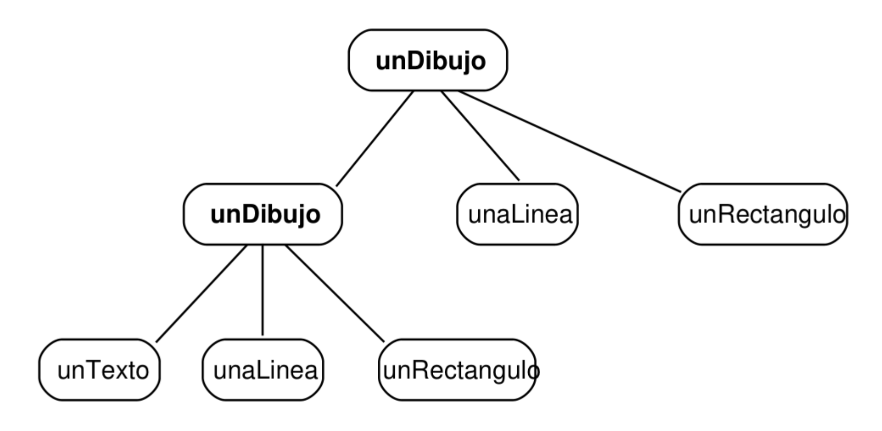
Las subclases Linea, Rectángulo y Texto (*véase* el diagrama de clases siguiente) definen objetos gráficos primitivos. Estas clases

implementan Dibujar para dibujar líneas, rectángulos y texto, respectivamente. Como los gráficos primitivos no tienen gráficos hijos, ninguna de estas clases implementa operaciones relacionadas con los hijos.



La clase Dibujo define una agregación de objetos Gráfico. Dibujo implementa Dibujar para que llame al Dibujar de sus hijos, y añade operaciones relacionadas con los hijos. Como la interfaz de Dibujo se ajusta a la interfaz de Gráfico, los objetos Dibujo pueden componer recursivamente otros Dibujos.

El siguiente diagrama muestra una típica estructura de objetos compuestos recursivamente por otros objetos Gráfico compuestos:

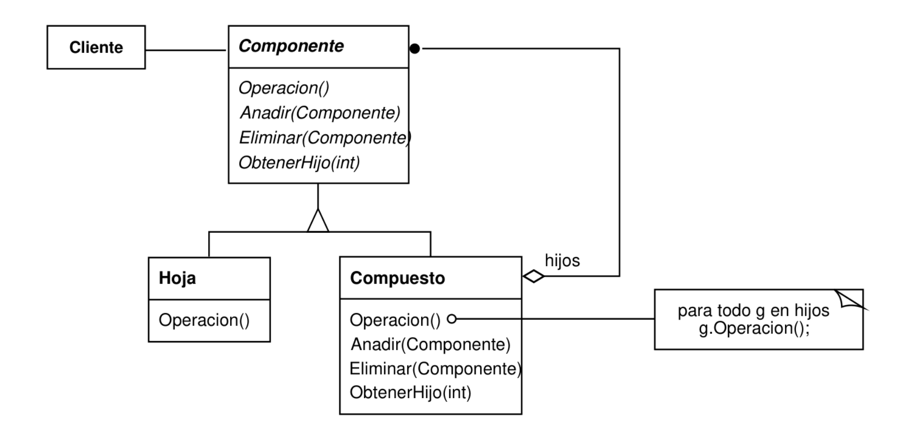


APLICABILIDAD

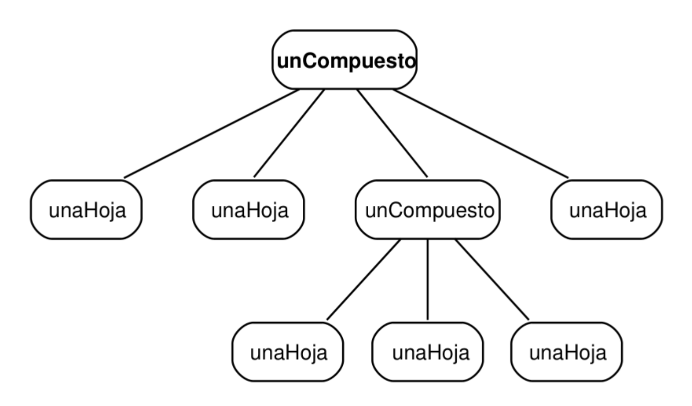
Use el patrón Composite cuando

- quiera representar jerarquías de objetos parte-todo.
- quiera que los clientes sean capaces de obviar las diferencias entre composiciones de objetos y los objetos individuales. Los clientes tratarán a todos los objetos de la estructura compuesta de manera uniforme.

ESTRUCTURA



Una estructura de objetos Compuestos típica puede parecerse a esto:



PARTICIPANTES

- **Componente** (Gráfico)
  - declara la interfaz de los objetos de la composición.

    - implementa el comportamiento predeterminado de la interfaz que es común a todas las clases.
    - declara una interfaz para acceder a sus componentes hijos y gestionarlos.
    - (opcional) define una interfaz para acceder al padre de un componente en la estructura recursiva y, si es necesario, la implementa.
- **Hoja** (Rectángulo, Linea, Texto, etc.)
    - representa objetos hoja en la composición. Una hoja no tiene hijos.
    - define el comportamiento de los objetos primitivos de la composición.
- **Compuesto** (Dibujo)
    - define el comportamiento de los componentes que tienen hijos.
    - almacena componentes hijos.
    - implementa las operaciones de la interfaz Componente relacionadas con los hijos.
- **Cliente**
    - manipula objetos en la composición a través de la interfaz Componente.

COLABORACIONES

Los Clientes usan la interfaz de la clase Componente para interactuar con los objetos de la estructura compuesta. Si el recipiente es una Hoja, la petición se trata correctamente. Si es un Compuesto, normalmente redirige las peticiones a sus componentes hijos, posiblemente realizando operaciones adicionales antes o después.

CONSECUENCIAS

El patrón Composite

- define jerarquías de clases formadas por objetos primitivos y compuestos. Los objetos primitivos pueden componerse en otros objetos más complejos, que a su vez pueden ser

compuestos, y así de manera recurrente. Allí donde el código espere un objeto primitivo, también podrá recibir un objeto compuesto.

- simplifica el cliente. Los clientes pueden tratar uniformemente a las estructuras compuestas y a los objetos individuales. Los clientes normalmente no conocen (y no les debería importar) si están tratando con una hoja o con un componente compuesto. Esto simplifica el código del cliente, puesto que evita tener que escribir funciones con instrucciones if anidadas en las clases que definen la composición.
- facilita añadir nuevos tipos de componentes. Si se definen nuevas subclases Compuesto u Hoja, éstas funcionarán automáticamente con las estructuras y el código cliente existentes. No hay que cambiar los clientes para las nuevas clases Componente.
- puede hacer que un diseño sea demasiado general. La desventaja de facilitar añadir nuevos componentes es que hace más difícil restringir los componentes de un compuesto. A veces queremos que un compuesto sólo tenga ciertos componentes. Con el patrón Composite, no podemos confiar en el sistema de tipos para que haga cumplir estas restricciones por nosotros. En vez de eso, tendremos que usar comprobaciones en tiempo de ejecución.

IMPLEMENTACIÓN

Hay muchas cuestiones a tener en cuenta al implementar el patrón Composite:

1. *Referencias explícitas al padre*. Mantener referencias de los componentes hijos a sus padres puede simplificar el recorrido y la gestión de una estructura compuesta. La referencia al padre facilita ascender por la estructura y borrar un componente. Las referencias al padre también ayudan a implementar el patrón Chain of Responsibility (205).

   El lugar habitual donde definir la referencia al padre es en

la clase Componente. Las clases Hoja y Compuesto pueden heredar la referencia y las operaciones que la gestionan.

Con referencias al padre, es esencial mantener el invariante de que todos los hijos de un compuesto tienen como padre al compuesto que a su vez los tiene a ellos como hijos. El modo más fácil de garantizar esto es cambiar el padre de un componente *sólo* cuando se añade o se elimina a éste de un compuesto. Si se puede implementar una vez en las operaciones Añadir y Eliminar de la clase Compuesto entonces puede ser heredado por todas las subclases, conservando automáticamente el invariante.

2. *Compartir componentes*. Muchas veces es útil compartir componentes, por ejemplo para reducir los requisitos de almacenamiento. Pero cuando un componente no puede tener más de un padre, compartir componentes se hace más difícil.

   Una posible solución es que los hijos almacenen múltiples padres. Pero eso puede llevarnos a ambigüedades cuando se propaga una petición hacia arriba en la estructura. El patrón Flyweight (179) muestra cómo adaptar un diseño para evitar guardar los padres. Funciona en casos en los que los hijos pueden evitar enviar peticiones al padre externalizando parte de su estado, o todo.

3. *Maximizar la interfaz Componente*. Uno de los objetivos del patrón Composite es hacer que los clientes se despreocupen de las clases Hoja o Compuesto que están usando. Para conseguirlo, la clase Componente debería definir tantas operaciones comunes a las clases Compuesto y Hoja como sea posible. La clase Componente normalmente proporciona implementaciones predeterminadas para estas operaciones, que serán redefinidas por las subclases Hoja y Compuesto.

   No obstante, este objetivo a veces entra en conflicto con el principio de diseño de **jerarquías** de clases que dice que

una clase sólo debería definir operaciones que tienen sentido en sus subclases. Hay muchas operaciones permitidas por Componente que no parecen tener sentido en las clases Hoja. ¿Cómo puede Componente proporcionar una implementación predeterminada para ellas?

A veces un poco de creatividad muestra cómo una operación que podría parecer que sólo tiene sentido en el caso de los Compuestos puede implementarse para todos los Componentes, moviéndola a la clase Componente. Por ejemplo, la interfaz para acceder a los hijos es una parte fundamental de la clase Compuesto, pero no de las clases Hoja. Pero si vemos a una Hoja como un Componente que *nunca* tiene hijos, podemos definir una operación predeterminada en la clase Componente para acceder a los hijos que nunca *devuelve* ningún hijo. Las clases Hoja pueden usar esa implementación predeterminada, pero las clases Compuesto la reimplementarán para que devuelva sus hijos.

Las operaciones de gestión de los hijos son más problemáticas, y se tratan en el siguiente punto.

4. *Declarar las operaciones de gestión de los hijos*. Aunque la clase Compuesto *implementa* las operaciones Añadir y Eliminar para controlar los hijos, un aspecto importante del patrón Compuesto es qué clases declaran estas operaciones en la jerarquía de clases Compuesto. ¿Deberíamos declarar estas operaciones en el Componente y hacer que tuvieran sentido en las clases Hoja, o deberíamos declararlas y definirlas sólo en Compuesto y sus subclases?

   La decisión implica un equilibrio entre seguridad y transparencia:

- Definir la interfaz de gestión de los hijos en la raíz de la jerarquía de clases nos da transparencia, puesto que podemos tratar a todos los componentes de manera uniforme. Sin

embargo, sacrifica la seguridad, ya que los clientes pueden intentar hacer cosas sin sentido, como añadir y eliminar objetos de las hojas.

- Definir la gestión de los hijos en la clase Compuesto nos proporciona seguridad, ya que cualquier intento de añadir o eliminar objetos de las hojas será detectado en tiempo de compilación en un lenguaje estáticamente tipado, como C++. Pero perdemos transparencia, porque las hojas y los compuestos tienen interfaces diferentes.

En este patrón hemos dado más importancia a la transparencia que a la seguridad. Si se opta por la seguridad, habrá ocasiones en las que perderemos información sobre el tipo y tendremos que convertir un componente en un compuesto. ¿Cómo podemos hacerlo sin recurrir a una conversión que no sea segura con respecto al tipo?

Una posibilidad es declarar una operación Compuesto* ObtenerCompuesto () en la clase Componente. El Componente proporciona una operación por omisión que devuelve un puntero nulo. La clase Compuesto redefine esta operación para devolverse a sí misma a través de su puntero this:

```
class Compuesto;

class Componente {
public:
      //…
      virtual                 Compuesto*
ObtenerCompuesto() { return 0; }
};

class Compuesto : public Componente
{
public:
      void Anadir(Componente*);
      // …
      virtual                 Compuesto*
ObtenerCompuesto() { return this; }
};
```

```
class Hoja : public Componente {
     // …
};
```

ObtenerCompuesto nos permite consultara un componente para ver si es un compuesto. Podemos ejecutar Añadir y Eliminar con seguridad sobre el compuesto que devuelve.

```
Compuesto*    unCompuesto    =    new
Compuesto;
Hoja* unaHoja = new Hoja;

Componente* unComponente;
Compuesto* prueba;

unComponente = unCompuesto;
if     (prueba     =     unComponente-
>ObtenerCompuesto()) {
     prueba->Anadir(new Hoja);
}

unComponente = unaHoja;

if     (prueba     =     unComponente-
>ObtenerCompuesto()) {
     prueba->Anadir(new Hoja); // no
añadirá a una hoja
}
```

Se pueden hacer comprobaciones similares para un Compuesto usando la construcción de C++ dynamic_cast.

Por supuesto, el problema aquí es que no tratamos a todos los componentes de manera uniforme. Tenemos que volver a realizar comprobaciones para diferentes tipos antes de emprender la acción apropiada.

La única forma de proporcionar transparencia es definir operaciones predeterminadas Añadir y Eliminar en Componente. Eso crea un nuevo problema: no hay modo de implementar Componente::Añadir sin introducir así mismo la posibilidad de que falle. Podríamos no hacer

nada, pero eso omite una consideración importante; es decir, un intento de añadir algo a una hoja probablemente esté indicando un error. En ese caso, la operación Anadir produce basura. Podríamos hacer que borrara su argumento, pero eso no es lo que los clientes esperan. Normalmente es mejor hacer que Anadir y Eliminar fallen de manera predeterminada (tal vez lanzando una excepción) si el componente no puede tener hijos o si el argumento de Eliminar no es un hijo del componente.

Otra posibilidad es cambiar ligeramente el significado de "eliminar". Si el componente mantiene una referencia al padre podríamos redefinir Componente::Eliminar para eliminarse a sí mismo de su padre. No obstante, sigue sin haber una interpretación con sentido para Añadir.

5. *¿Debería implementar el Componente una lista de Componentes*? Podríamos estar tentados de definir el conjunto de hijos como una variable de instancia de la clase Componente en la que se declaren las operaciones de acceso y gestión de los hijos. Pero poner el puntero al hijo en la clase base incurre en una penalización de espacio para cada hoja, incluso aunque una hoja nunca tenga hijos. Esto sólo merece la pena si hay relativamente pocos hijos en la estructura.

6. *Ordenación de los hijos*. Muchos diseños especifican una ordenación de los hijos de Compuesto. En el ejemplo del Gráfico, la ordenación puede reflejar el orden desde el frente hasta el fondo. Si los objetos Compuesto representan árboles de análisis, entonces las instrucciones compuestas pueden ser instancias de un Compuesto cuyos hijos deben estar ordenados de manera que reflejen el programa.

   Cuando la ordenación de los hijos es una cuestión a tener en cuenta, debemos diseñar las intertaces de acceso y gestión de hijos cuidadosamente para controlar la secuencia de hijos. El patrón Iterador (237) puede servirnos de guía.

7. *Caché para mejorar el rendimiento*. Si necesitamos recorrer composiciones o buscar en ellas con frecuencia, la clase Compuesto puede almacenar información sobre sus hijos que facilite el recorrido o la búsqueda. El Compuesto puede guardar resultados o simplemente información que le permita evitar parte del recorrido o de la búsqueda. Por ejemplo, la clase Dibujo del ejemplo de la sección de Motivación podría guardar la caja limítrofe de sus hijos. Mientras se dibuja o es seleccionado, esta caja previamente guardada permite que Dibujo no tenga que dibujar o realizar búsquedas cuando sus hijos no son visibles en la ventana actual.

   Los cambios en un componente requerirán invalidar la caché de sus padres. Esto funciona mejor cuando los componentes conocen a sus padres. Por tanto, si se va a usar almacenamiento caché se necesita definir una interfaz para decirle a los compuestos que su caché ya no es válida.

8. ¿*Quién debería borrar los componentes*? En lenguajes sin recolección de basura, normalmente es mejor hacer que un Compuesto sea el responsable de borrar sus hijos cuando es destruido. Una excepción a esta regla es cuando los objetos Hoja son inmutables y pueden por tanto ser compartidos.

9. ¿*Cuál es la mejor estructura de datos para almacenar los componentes*? Los objetos Compuesto pueden usar muchas estructuras de datos diferentes para almacenar sus hijos, incluyendo listas enlazadas, árboles, arrays y tablas de dispersión. La elección de la estructura de datos depende (como siempre) de la eficiencia. De hecho, ni siquiera es necesario usar una estructura de datos de propósito general. A veces los compuestos tienen una variable para cada hijo, aunque esto requiere que cada subclase de Compuesto implemente su propia interfaz de gestión. *Véase* el patrón Interpreter (225) para un ejemplo.

CÓDIGO DE EJEMPLO

Determinados equipos, como computadores y componentes estéreo, suelen estar organizados en jerarquías de parte-todo o de pertenencia. Por ejemplo, un chasis puede contener unidades y placas base, un bus puede contener tarjetas y un armario puede contener chasis, buses, etcétera. Dichas estructuras pueden modelarse de manera natural con el patrón Composite.

La clase Equipo define una interfaz para todos los equipos de la jerarquía de parte-todo.

```
class Equipo {
public:
     virtual ~Equipo();

     const char* Nombre() { return _nombre; }

     virtual Vatio Potencian;
     virtual Moneda PrecioNeto();
     virtual Moneda PrecioConDescuento();

     virtual void Anadir(Equipo*);
     virtual void Eliminar(Equipo*);
     virtual               Iterador<Equipo*>*
CrearIterador();
protected:
     Equipo(const char*);
private:
     const char* _nombre;
};
```

Equipo declara operaciones que devuelven los atributos de un equipo, como su consumo y coste. Las subclases implementan estas operaciones para determinados tipos de equipos. Equipo también declara una operación CrearIterador que devuelve un Iterador (*véase* el Apéndice C) para acceder a sus partes. La implementación predeterminada de esta operación devuelve un IteradorNulo, que itera sobre el conjunto vacío.

Las subclases de Equipo podrían incluir clases Hoja que representen unidades de disco, circuitos integrados e interruptores:

```
class Disquetera : public Equipo {
public:
```

```
        Disquetera(const char*);
        virtual ~Disquetera();

        virtual Vatio Potencia();
        virtual Moneda PrecioNeto();
        virtual Moneda PrecioConDescuento();
};
```

EquipoCompuesto es la clase base de los equipos que contienen otros equipos. Es también una subclase de Equipo.

```
class EquipoCompuesto : public Equipo {
public:
        virtual ~EquipoCompuesto();

        virtual Vatio Potencia();
        virtual Moneda PrecioNeto();
        virtual Moneda PrecioConDescuento();

        virtual void Anadir(Equipo*);
        virtual void Eliminar(Equipo*);
        virtual                Iterador<Equipo*>*
CrearIterador();

protected:
        EquipoCompuesto(const char*);
private:
        Lista<Equipo*> _equipo;
};
```

EquipoCompuesto define las operaciones para acceder a sus componentes y recorrerlos. Las operaciones Añadir y Eliminar insertan y borran equipos en la lista de equipos almacenados en el miembro _equipo. La operación CrearIterador devuelve un iterador (concretamente, una instancia de IteradorLista) para recorrer la lista.

Una implementación predeterminada de PrecioNeto podría usar CrearIterador para sumar los precios netos de los equipos que lo componen[37]:

```
Moneda EquipoCompuesto::PrecioNeto () {
```

```
    Iterador<Equipo*>* i = CrearIterador();
    Moneda total = 0;

    for (i->Primero(); !i->HaTerminado(); i-
>Siguiente()) {
        total    +=    i->ElementoActual()-
>PrecioNeto();
    }
    delete i;
    return total;
}
```

Ahora podemos representar un chasis de computadora como una subclase de EquipoCompuesto llamada Chasis. Chasis hereda las operaciones relativas a los hijos de EquipoCompuesto.

```
class Chasis : public EquipoCompuesto {
public:
    Chasis(const char*);
    virtual ~Chasis();

    virtual Vatio Potencia();
    virtual Moneda PrecioNeto();
    virtual Moneda PrecioConDescuento();
};
```

Podemos definir otros contenedores de equipos tales como Armario y Bus de forma similar. Eso nos da todo lo necesario para ensamblar componentes en una computadora personal (bastante sencillo):

```
Armario* armario = new Armario ("Armario de
PC");
Chasis* chasis = new Chasis ("Chasis de PC");

armario->Anadir(chasis);

Bus* bus = new Bus("Bus MCA");
bus->Anadir(new Tarjeta("Token Ring de 16 Mbs
"));

chasis->Anadir(bus);
chasis->Anadir(new Disquetera("Disquetera de
3,5 pulgadas"));
```

```
cout « "El precio neto es " « chasis-
>PrecioNeto() « endl;
```

USOS CONOCIDOS

Se pueden encontrar ejemplos del patrón Composite en casi todos los sistemas orientados a objetos. La clase Vista original del Modelo/Vista/Controlador de Smalltalk [KP88] era un Compuesto, y prácticamente todos los toolkits o frameworks de interfaces de usuario han seguido sus pasos, incluyendo ET++ (con VObjects [WGM88]) e Interviews (Styles [LCI+92], Graphics [VL88] y Glyphs [CL90]). Merece la pena destacar que la Vista original del Modelo/Vista/Controlador tenía un conjunto de subvistas; en otras palabras, la clase View era tanto la clase Componente como la Compuesto. La versión 4.0 de Smalltalk-80 revisó el Modelo/Vista/Controlador con una clase VisualComponent que tenía como subclases View y CompositeView.

El framework para compiladores de Smalltalk RTL (JML92] hace un uso intensivo del patrón Composite. RTLExpression es una clase Componente para árboles de análisis. Tiene subclases, tales como Binary Expression, que contienen objetos RTLExpression como hijos. Dichas clases definen una estructura compuesta para árboles de análisis. RegisterTransfer es la clase Componente para un programa en la forma intermedia de Single Static Assignment (SSA). Las subclases Hoja de RegisterTransfer definen diferentes asignaciones estáticas, como

- asignaciones primitivas que realizan una operación en dos registros y asignan el resultado a un tercero:
- una asignación con un registro fuente pero sin registro destino, que indica que el registro se usa después del retomo de una rutina: y
- una asignación con un registro destino pero sin origen, lo que indica que al registro se le asigna un valor antes de que empiece la rutina.

Otra subclase, RegisterTransferSet, es una clase Compuesto que representa asignaciones que modifican varios registros a la vez.

Otro ejemplo de este patrón tiene lugar en el dominio de las

finanzas, donde una cartera de acciones agrupa valores individuales. Se pueden permitir agregaciones complejas de valores implementando una cartera como un Compuesto que se ajusta a la interfaz de un valor individual [BE93].

El patrón Command (215) describe cómo se pueden componer y secuenciar objetos Orden con una clase Compuesto OrdenMacro.

PATRONES RELACIONADOS

Muchas veces se usa el enlace al componente padre para implementar el patrón Chain of Responsibility (205).

El patrón Decorator (161) suele usarse junto con el Composite. Cuando se usan juntos decoradores y compuestos, normalmente ambos tendrán una clase padre común. Por tanto, los decoradores tendrán que admitir la interfaz Componente con operaciones como Añadir, Eliminar y ObtenerHijo.

El patrón Flyweight (179) permite compartir componentes, si bien en ese caso éstos ya no pueden referirse a sus padres.

Se puede usar el patrón Iterator (237) para recorrer las estructuras definidas por el patrón Composite.

El patrón Visitor (305) localiza operaciones y comportamiento que de otro modo estaría distribuido en varias clases Compuesto y Hoja.

Estructural de Objetos

DECORATOR
(Decorador)

PROPÓSITO

Asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

TAMBIÉN CONOCIDO COMO

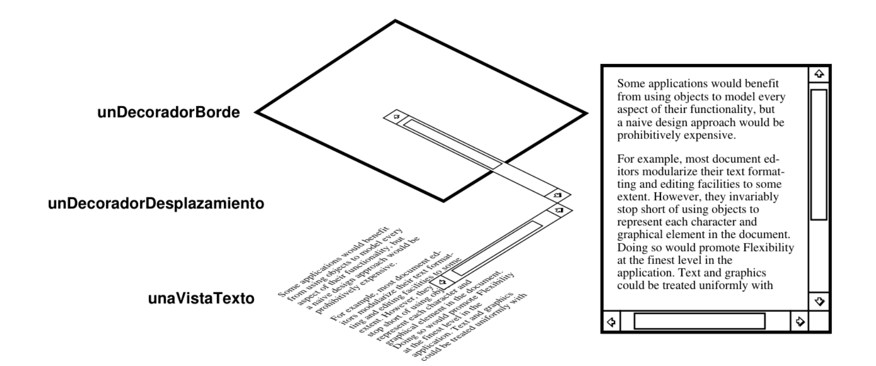
Wrapper (Envoltorio)

MOTIVACIÓN

A veces queremos añadir responsabilidades a objetos individuales en vez de a toda una clase. Por ejemplo, un toolkit de interfaces de usuario debería permitir añadir propiedades (como bordes) o comportamientos (como capacidad de desplazamiento) a cualquier componente de la interfaz de usuario.

Un modo de añadir responsabilidades es a través de la herencia. Heredar un borde de otra clase pondría un borde alrededor de todas las instancias de la subclase. Sin embargo, esto es inflexible, ya que la elección del borde se hace estáticamente. Un cliente no puede controlar cómo y cuándo decorar el componente con un borde.

Un enfoque más flexible es encerrar el componente en otro objeto que añada el borde. Al objeto confinante se le denomina **decorador**. El decorador se ajusta a la interfaz del componente que decora de manera que su presencia es transparente a sus clientes. El decorador reenvía las peticiones al componente y puede realizar acciones adicionales (tales como dibujar un borde) antes o después del reenvío. Dicha transparencia permite anidar decoradores recursivamente, permitiendo así un número ilimitado de responsabilidades añadidas.

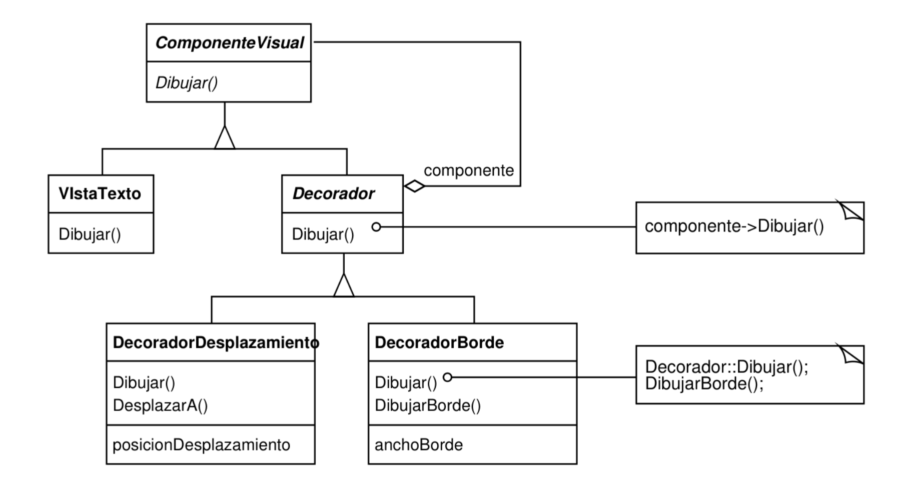


Por ejemplo, supongamos un objeto VistaTexto que muestra texto en una ventana. VistaTexto no tiene barras de desplazamiento de manera predeterminada, ya que puede que no sean siempre necesarias. Cuando las necesitemos, podemos usar un DecoradorDesplazamiento para añadirlas. Supongamos que queremos añadir un borde negro, grueso, alrededor de VistaTexto. Podemos usar un DecoradorBorde para añadir también el borde. Basta con componer los decoradores con VistaTexto para producir el resultado deseado.

El siguiente diagrama de objetos muestra cómo componer un objeto VistaTexto con objetos DecoradorBorde y DecoradorDesplazamiento para producir una vista de texto con borde y desplazamiento:



Las clases DecoradorDesplazamiento y DecoradorBorde son subclases de Decorador, una clase abstracta para componentes visuales que decoran otros componentes visuales.



Componente Visual es la clase abstracta de los objetos visuales. Define su interfaz para dibujarse y para el manejo de eventos. Nótese cómo la clase Decorador simplemente redirige las peticiones para que se dibuje su componente, y cómo las subclases de Decorador pueden extender dicha operación.

Las subclases de Decorador son libres de añadir operaciones para determinadas funcionalidades. Por ejemplo, la operación Desplazar A de DecoradorDesplazamiento permite que otros objetos desplacen la interfaz si saben que la interfaz incluye un objeto DecoradorDesplazamiento. Lo importante de este patrón es que permite que los decoradores aparezcan en cualquier lugar en el que pueda ir un Componente Visual. De esa forma los clientes generalmente no pueden distinguir entre un componente decorado y otro que no lo está, por lo que no dependen en absoluto de la decoración.

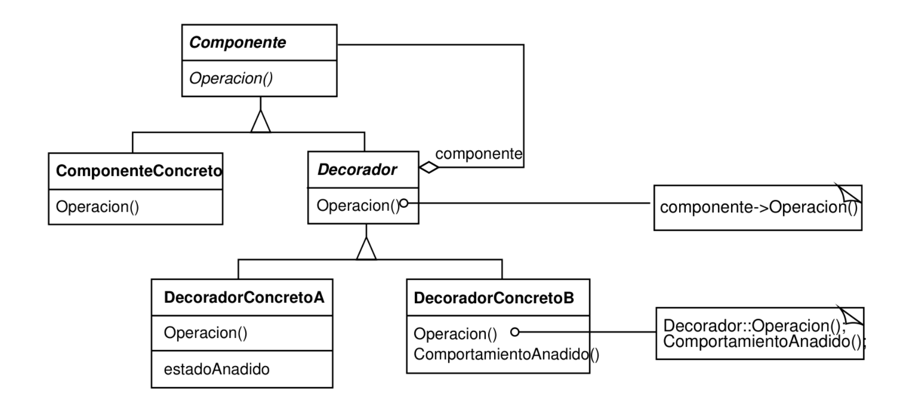
APLICABILIDAD

Use el Decorador

- para añadir objetos individuales de forma dinámica y transparente, es decir, sin afectar a otros objetos.
- para responsabilidades que pueden ser retiradas.
- cuando la extensión mediante la herencia no es viable. A veces es posible tener un gran número de extensiones independientes, produciéndose una explosión de subclases

para permitir todas las combinaciones. O puede ser que una definición de una clase esté oculta o que no esté disponible para ser heredada.

ESTRUCTURA



PARTICIPANTES

- **Componente** (ComponenteVisual)
  - define la interfaz para objetos a los que se puede añadir responsabilidades dinámicamente.
- **ComponenteConcreto** (VistaTexto)
  - define un objeto al que se pueden añadir responsabilidades adicionales.
- **Decorador**
  - mantiene una referencia a un objeto Componente y define una interfaz que se ajusta a la interfaz del Componente.
- **DecoradorConcreto** (DecoradorBorde, DecoradorDesplazamiento)
  - añade responsabilidades al componente.

COLABORACIONES

El Decorador redirige peticiones a su objeto Componente.

Opcionalmente puede realizar operaciones adicionales antes y después de reenviar la petición.

CONSECUENCIAS

El patrón Decorador tiene al menos dos ventajas y dos inconvenientes fundamentales:

1. *Más flexibilidad que la herencia estática*. El patrón Decorador proporciona una manera más flexible de añadir responsabilidades a los objetos que la que podía obtenerse a través de la herencia (múltiple) estática. Con los decoradores se pueden añadir y eliminar responsabilidades en tiempo de ejecución simplemente poniéndolas y quitándolas. Por el contrario, la herencia requiere crear una nueva clase para cada responsabilidad adicional (como VistaTextoDesplazableConBorde o VistaTextoConBorde). Esto da lugar a muchas clases diferentes e incrementa la complejidad de un sistema. Por otro lado, proporcionar diferentes clases Decorador para una determinada clase Componente permite mezclar responsabilidades.

   Los Decoradores también facilitan añadir una propiedad dos veces. Por ejemplo, para dar un borde doble a VistaTexto, basta con añadir dos objetos DecoradorBorde. Heredar dos veces de una clase Borde resulta, cuando menos, propenso a errores.

2. *Evita clases cargadas de funciones en la parte de arriba de la jerarquía*. El Decorador ofrece un enfoque para añadir responsabilidades que consiste en pagar sólo por aquello que se necesita. En vez de tratar de permitir todas las funcionalidades inimaginables en una clase compleja y adaptable, podemos definir primero una clase simple y añadir luego funcionalidad incrementalmente con objetos Decorador. La funcionalidad puede obtenerse componiendo partes simples. Como resultado, una aplicación no necesita pagar por características que no usa. También resulta fácil definir nuevos tipos de

Decoradores independientemente de las clases de objetos de las que hereden, incluso para extensiones que no hubieran sido previstas. Extender una clase compleja tiende a exponer detalles no relacionados con las responsabilidades que estamos añadiendo.

3. *Un decorador y su componente no son idénticos*. Un decorador se comporta como un revestimiento transparente. Pero desde el punto de vista de la identidad de un objeto, un componente decorado no es idéntico al componente en sí. Por tanto, no deberíamos apoyarnos en la identidad de objetos cuando estamos usando decoradores.

4. *Muchos objetos pequeños*. Un diseño que usa el patrón Decorator suele dar como resultado sistemas formados por muchos objetos pequeños muy parecidos. Los objetos sólo se diferencian en la forma en que están interconectados, y no en su clase o en el valor de sus variables. Aunque dichos sistemas son fáciles de adaptar por parte de quienes los comprenden bien, pueden ser difíciles de aprender y de depurar.

IMPLEMENTACIÓN

Hay que tener en cuenta varias cuestiones al aplicar el patrón Decorator:

1. *Concordancia de interfaces*. La interfaz de un objeto decorador debe ajustarse a la interfaz del componente que decora. Las clases DecoradorConcreto deben por tanto heredar de una clase común (al menos en C++).

2. *Omisión de la clase abstracta Decorador*. No hay necesidad de definir una clase abstracta Decorador cuando sólo necesitamos añadir una responsabilidad. Eso es lo que suele ocurrir cuando estamos tratando con una jerarquía de clases existente y no diseñando una nueva. En ese caso, podemos obtener la responsabilidad del Decorador reenviando peticiones al componente en el

DecoradorConcreto.

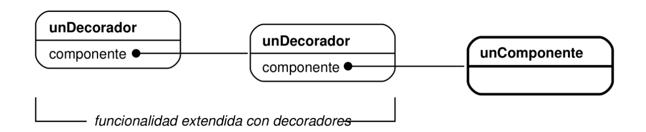
3. *Mantener ligeras las clases Componente*. Para garantizar una interfaz compatible, los componentes y los decoradores deben descender de una clase Componente común. Es importante que esta clase común se mantenga ligera; es decir, debería centrarse en definir una interfaz, no en guardar datos. La definición de cómo se representan los datos debería delegarse en las subclases; de no ser así, la complejidad de la clase Componente puede hacer que los decoradores sean demasiado pesados como para usar una gran número de ellos. Poner mucha funcionalidad en el Componente también incrementa la probabilidad de que las subclases concretas estén pagando por características que no necesitan.

4. *Cambiar la piel de un objeto en vez de sus tripas*. Podemos pensar en un decorador como un revestimiento de un objeto que cambia su comportamiento. Una alternativa es cambiar las interioridades del objeto. El patrón Strategy (289) es un buen ejemplo de patrón para cambiar las tripas. Las estrategias son una mejor elección en aquellas situaciones en las que la clase Componente es intrínsecamente pesada, lo que hace que el patrón Decorator sea demasiado costoso de aplicar. En el patrón Strategy, el componente delega parte de su responsabilidad en un objeto estrategia aparte. El patrón Strategy nos permite alterar o extender la funcionalidad del componente cambiando el objeto estrategia.

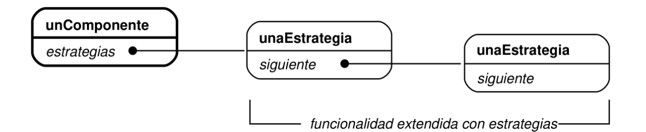
   Por ejemplo, podemos permitir diferentes estilos de bordes haciendo que el componente delegue el dibujado del borde en un objeto Borde aparte. El objeto Borde es un objeto Estrategia que encapsula un algoritmo para dibujar bordes. Al ampliar el número de estrategias de una a una lista ilimitada conseguimos el mismo efecto que anidando recursivamente decoradores. Por ejemplo, en MacApp 3.0 [App89] y Bedrock [Sym93a] los componentes gráficos (denominados "vistas") mantienen una lista de objetos

“adorno” que pueden añadir adornos adicionales, como bordes a un componente vista. Si una vista tiene algunos adornos, les da una posibilidad de dibujar adornos adicionales. MacApp y Bedrock deben usar este enfoque debido a que la clase View es pesada, por lo que sería demasiado costoso usarla sólo para añadir un borde.

Puesto que el patrón Decorator sólo cambia la parte exterior de un objeto, el componente no tiene que saber nada de sus decoradores; es decir, los decoradores son transparentes para el componente:



Con estrategias, el componente conoce sus posibles extensiones. Por tanto tiene que referenciar y mantener las estrategias correspondientes:



El enfoque basado en estrategias puede requerir modificar el componente para permitir nuevas extensiones. Por otro lado, una estrategia puede tener su propia interfaz especializada, mientras que la interfaz de un decorador debe ajustarse a la del componente. Una estrategia para dibujar un borde, por ejemplo, sólo necesita definir la interfaz para dibujar el borde (DibujarBorde, ObtenerAncho, etc.), lo que significa que la estrategia puede ser ligera incluso aunque la clase componente sea pesada.

MacApp y Bedrock usan este enfoque para algo más que simplemente adornar vistas. También la usan para aumentar el comportamiento de manejo de eventos de los objetos. En ambos sistemas, una vista mantiene una lista de objetos de “comportamiento” que pueden modificare

interceptar eventos. La vista da la posibilidad de manejar el evento a cada uno de los objetos de comportamiento registrados antes que a los no registrados. Podemos decorar una vista para que admita el manejo de eventos, por ejemplo, registrando un objeto comportamiento que intercepte y maneje los eventos de teclado.

CÓDIGO DE EJEMPLO

El código siguiente muestra cómo implementar decoradores de interfaz de usuario en C++. Supondremos que hay una clase Componente llamada ComponenteVisual.

```
class ComponenteVisual {
public:
     ComponenteVisual();

     virtual void Dibujar();
     virtual void CambiarTamano();
     // …
};
```

Definiremos una subclase de ComponenteVisual llamada Decorador, de la cual heredaremos para obtener diferentes decoraciones.

```
class Decorador : public ComponenteVisual {
public:
     Decorator(ComponenteVisual*);

     virtual void Dibujar();
     virtual void CambiarTamano();
     // …
private:
     ComponenteVisual* _componente;
};
```

Decorador decora el ComponenteVisual referenciado por la variable de instancia _componente, la cual es inicializada en el constructor. Para cada operación de la interfaz ComponenteVisual, Decorador define una implementación predeterminada que pasa la petición a

_componente:

```
void Decorador::Dibujar () {
     _componente->Dibujar();
}

void Decorador::CambiarTamano () {
     _componente->CambiarTamano();
}
```

Las subclases de Decorador definen decoraciones concretas. Por ejemplo, la clase DecoradorBorde añade un borde a su componente. DecoradorBorde es una subclase de Decorador que redefine la operación Dibujar para dibujar el borde. DecoradorBorde también define una operación privada auxiliar DibujarBorde que se encarga de dibujarlo. La subclase hereda de Decorador la implementación de todas las otras operaciones.

```
class DecoradorBorde : public Decorator {
public:
     DecoradorBorde(ComponenteVisual*,    int
anchoBorde);

     virtual void Dibujar();
private:
     void DibujarBorde(int);
private:
     int _ancho;
};

void DecoradorBorde::Dibujar () {
     Decorator::Dibujar();
     DibujarBorde(_ancho);
}
```

Aquí vendría una implementación parecida para DecoradorDesplazamiento y DecoradorSombra, que añadirían capacidades de desplazamiento y sombra a un componente visual.

Ahora podemos combinar instancias de estas clases para proporcionar diferentes decoraciones. El código siguiente ilustra

cómo podemos usar decoradores para crear una VistaTexto desplazable y con borde.

En primer lugar, necesitamos un modo de poner un objeto visual en un objeto ventana. Supondremos que nuestra clase Ventana proporciona una operación EstablecerContenido para este propósito:

```
void             Ventana::EstablecerContenido
(ComponenteVisual* contenido) {
     // …
}
```

Ahora podemos crear la vista de texto y una ventana donde ponerla:

```
Ventana* ventana = new Ventana;
VistaTexto* VistaTexto = new VistaTexto;
```

VistaTexto es un ComponenteVisual, lo que nos permite ponerla en la ventana:

```
ventana->EstablecerContenido(vistaTexto);
```

Pero queremos una VistaTexto con borde y que se pueda desplazar, por lo que hemos de decorarla de manera apropiada antes de ponerla en la ventana.

```
ventana->EstablecerContenido(
     new DecoradorBorde(
          new
DecoradorDesplazamiento(vistaTexto), 1
     )
);
```

Dado que Ventana accede a su contenido a través de la interfaz ComponenteVisual, no se percata de la presencia del decorador.

Nosotros, como clientes, todavía podemos utilizar la vista de texto si tenemos que interactuar con ella directamente, por ejemplo cuando necesitamos invocar operaciones que no son parte de la interfaz de ComponenteVisual. Los clientes que se basan en la identidad del componente también deberían referirse a ella directamente.

USOS CONOCIDOS

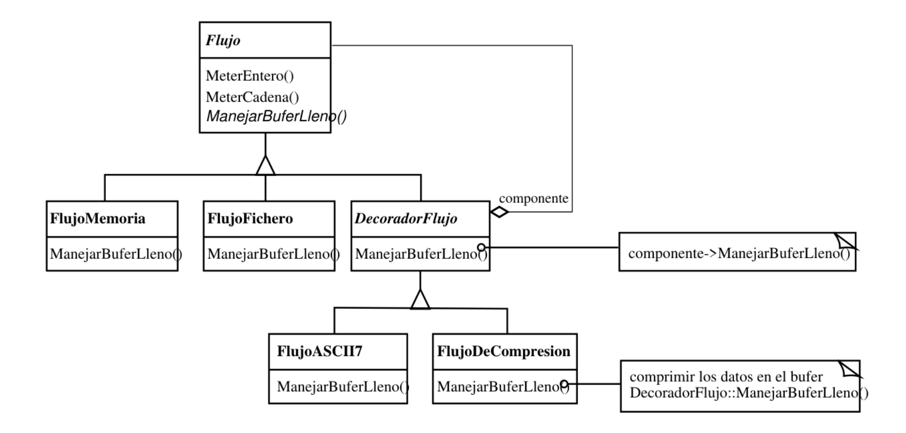
Muchos toolkits de interfaces de usuario orientados a objetos usan decoradores para añadir adornos gráficos a los útiles[38]. Algunos ejemplos son Interviews [LVC98, LCI+92], ET++ [WGM88] y la biblioteca de clases ObjectWorksVSmalltalk [Par90]. Aplicaciones más exóticas del patrón Decorator son el DebuggingGlyph de Interviews y el PassivityWrapper de ParcPlace Smalltalk. Un DebuggingGlyph muestra información de depuración antes y después de reenviar una petición a su componente para que se ubique en pantalla. Esta información de traza puede usarse para analizar y depurar el comportamiento de cómo se disponen los objetos en una composición compleja. El PassivityWrapper puede activar o desactivar las interacciones del usuario con el componente.

Pero el patrón Decorator no se limita en absoluto a las interfaces gráficas de usuario, como demuestra el siguiente ejemplo (basado en las clases de flujos de ET++ [WGM88]).

Los flujos son una abstracción fundamental en la mayoría de mecanismos de E/S. Un flujo puede proporcionar una interfaz para convertir objetos en una secuencia de bytes o caracteres. Eso nos permite transcribir un objeto a un fichero o a una cadena en memoria para recuperarlo posteriormente. Un modo sencillo de llevar esto a cabo es definiendo una clase abstracta Flujo con subclases como FlujoMemoria y FlujoFichero. Pero supongamos que también queremos poder hacer lo siguiente:

- Comprimir el flujo de datos usando diferentes algoritmos de compresión (run-length encoding, Lempel-Ziv, etc.).
- Reducir el flujo de datos a caracteres ASCII de 7 bits, de manera que puedan transmitirse sobre un canal de comunicación ASCII.

El patrón Decorador nos ofrece un modo elegante de añadir estas responsabilidades a los flujos. El siguiente diagrama muestra una solución al problema:



La clase abstracta Flujo mantiene un búfer interno y proporciona operaciones para almacenar datos en el flujo (MeterEntero, MeterCadena). Cuando se llene el búfer, Flujo llamará a la operación ManejarBuferLleno, que lleva a cabo la transferencia real de datos. La versión FlujoFichero de esta operación la redefine para transferir el búfer a un fichero.

La clase fundamental aquí es DecoradorFlujo, que mantiene una referencia a un componente flujo y le reenvía peticiones. Las subclases de DecoradorFlujo redefinen ManejarBuferLleno y llevan a cabo acciones adicionales llamando a la operación ManejarBuferLleno de DecoradorFlujo.

Por ejemplo, la subclase FlujoDeCompresion comprime los datos, y FlujoASC117 convierte los datos en ASCII de 7 bits. Ahora, para crear un FlujoFichero que comprima sus datos y convierta los datos binarios comprimidos en ASCII de 7 bits, decoramos un FlujoFichero con un FlujoDeCompresion y un FlujoASCII7:

```
Flujo* unFlujo = new FlujoCompresion(
     new FlujoASCII7(
          new
FlujoFichero("unNombreDeFichero")
     )
);
```

```
unFlujo->MeterEntero(12);
unFlujo->MeterCadena ("unaCadena");
```

PATRONES RELACIONADOS

Adapter (131): un decorador se diferencia de un adaptador en que el decorador sólo cambia las responsabilidades de un objeto, no su interfaz, mientras que un adaptador le da a un objeto una interfaz completamente nueva.

Composite (151): podemos ver a un decorador como un compuesto degenerado que sólo tiene un componente. No obstante, un decorador añade responsabilidades adicionales —no está pensado para la agregación de objetos—.

Strategy (289): un decorador permite cambiar el exterior de un objeto; una estrategia permite cambiar sus tripas. Son dos formas alternativas de modificar un objeto.

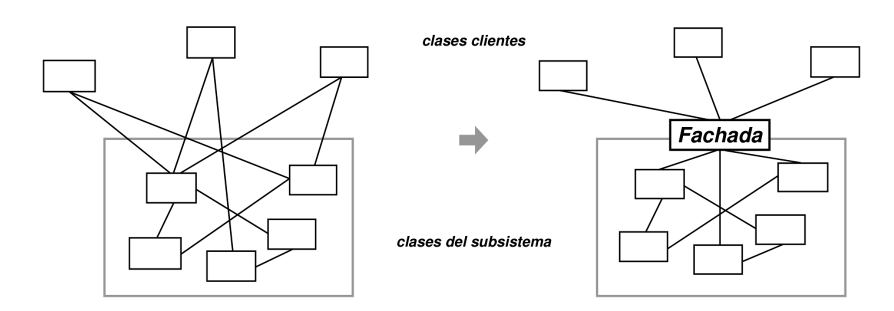
FACADE
(Fachada)

PROPÓSITO

Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.

MOTIVACIÓN

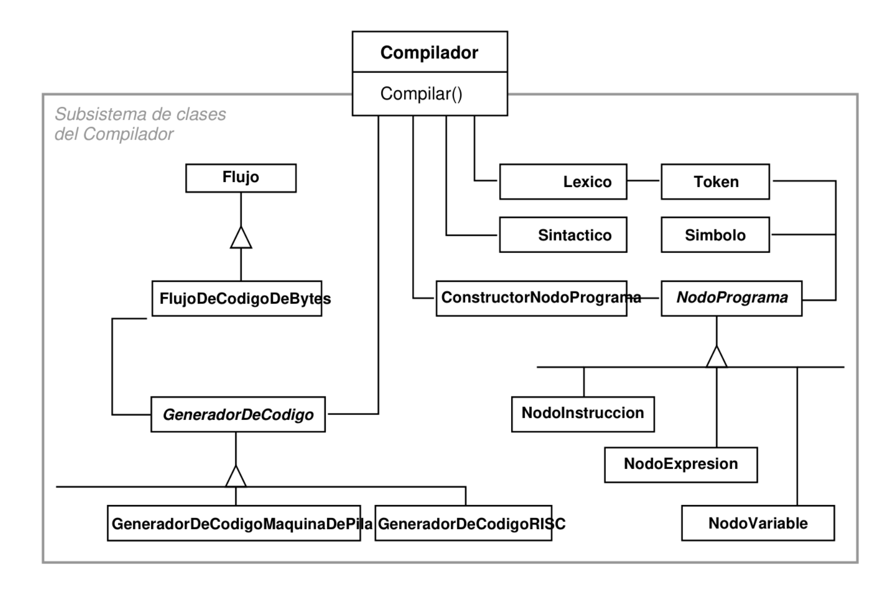
Estructurar un sistema en subsistemas ayuda a reducir la complejidad. Un típico objetivo de diseño es minimizar la comunicación y dependencias entre subsistemas. Un modo de lograr esto es introduciendo un objeto **fachada** que proporcione una interfaz única y simplificada para los servicios más generales del subsistema.



Pensemos, por ejemplo, en un entorno de programación que permita a las aplicaciones acceder a su subsistema de compilación. Este subsistema contendrá clases tales como Léxico, Sintáctico, Nodo-Programa, FlujoDeCodigoBinario y ConstructorNodoPrograma que implementan el compilador. Algunas aplicaciones especializadas podrían necesitar acceder a estas clases directamente, pero la mayoría de los clientes de un compilador no suelen preocuparse de detalles como el análisis sintáctico y la generación

de código, sino que simplemente quieren compilar un código determinado. Para estas aplicaciones, las potentes interfaces de bajo nivel del subsistema de compilación sólo complicarían su labor.

Para proporcionar una interfaz de más alto nivel que aísle a estas clases de los clientes el subsistema de compilación también incluye una clase Compilador. Esta clase define una interfaz uniforme para la funcionalidad del compilador. La clase Compilador funciona como una fachada: ofrece a los clientes una interfaz única y simple para el subsistema de compilación. Esta clase aglutina las clases que implementan la funcionalidad del compilador sin ocultarlas por completo. La fachada del compilador facilita la vida a la mayoría de los programadores, sin ocultar la funcionalidad de más bajo nivel para aquellos pocos que la necesiten.



APLICABILIDAD

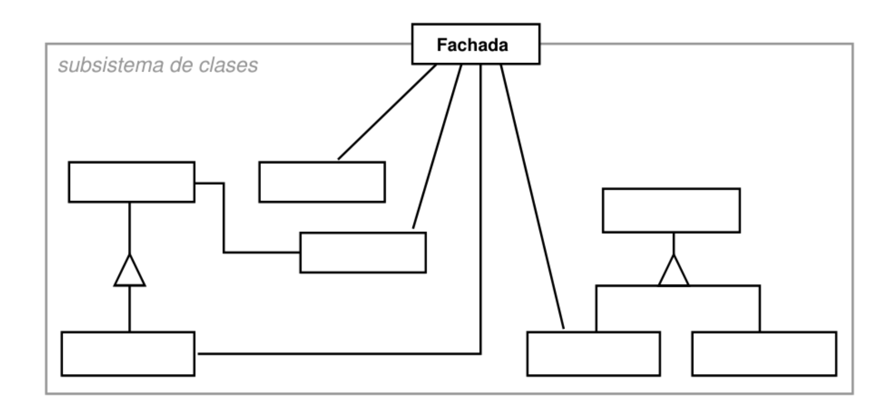
Usaremos el patrón Facade cuando

- queramos proporcionar una interfaz simple para un subsistema complejo. Los subsistemas suelen volverse más complicados a medida que van evolucionando. La mayoría de los patrones, cuando se aplican, dan como resultado más clases y más pequeñas. Esto hace que el subsistema sea más

reutilizable y fácil de personalizar, pero eso también lo hace más difícil de usar para aquellos clientes que no necesitan personalizarlo. Una fachada puede proporcionar, por omisión, una vista simple del subsistema que resulta adecuada para la mayoría de clientes. Sólo aquellos clientes, que necesitan más personalización necesitarán ir más allá de la fachada.

- haya muchas dependencias entre los clientes y las clases que implementan una abstracción. Se introduce una fachada para desacoplar el subsistema de sus clientes y de otros subsistemas, promoviendo así la independencia entre subsistemas y la portabilidad.
- queramos dividir en capas nuestros subsistemas. Se usa una fachada para definir un punto de entrada en cada nivel del subsistema. Si éstos son dependientes, se pueden simplificar las dependencias entre ellos haciendo que se comuniquen entre sí únicamente a través de sus fachadas.

ESTRUCTURA



PARTICIPANTES

- **Fachada** (Compilador)
  - sabe qué clases del subsistema son las responsables ante una petición.
  - delega las peticiones de los clientes en los objetos

apropiados del subsistema.

- **clases del subsistema** (Léxico, Sintáctico, NodoPrograma, etc.)
    - implementan la funcionalidad del subsistema.
    - realizan las labores encomendadas por el objeto Fachada.
    - no conocen a la fachada; es decir, no tienen referencias a ella.

COLABORACIONES

- Los clientes se comunican con el subsistema enviando peticiones al objeto Fachada, el cual las reenvía a los objetos apropiados del subsistema. Aunque son los objetos del subsistema los que realizan el trabajo real, la fachada puede tener que hacer algo de trabajo para pasar de su interfaz a las del subsistema.
- Los clientes que usan la fachada no tienen que acceder directamente a los objetos del subsistema.

CONSECUENCIAS

El patrón Facade proporciona las siguientes ventajas:

1. Oculta a los clientes los componentes del subsistema, reduciendo así el número de objetos con los que tratan los clientes y haciendo que el subsistema sea más fácil de usar.

2. Promueve un débil acoplamiento entre el subsistema y sus clientes. Muchas veces los componentes de un subsistema están fuertemente acoplados. Un acoplamiento débil nos permite modificar los componentes del subsistema sin que sus clientes se vean afectados. Las fachadas ayudan a estructurar en capas un sistema y las dependencias entre los objetos. También pueden eliminar dependencias complejas o circulares. Esto puede ser una consecuencia importante cuando el cliente y el subsistema se implementan por separado.

En los grandes sistemas software es vital reducir las dependencias de compilación. Queremos ahorrar tiempo minimizando la recompilación cuando cambien las clases del subsistema. Reducir las dependencias de compilación con fachadas puede limitar la recompilación necesaria para un pequeño cambio en un subsistema importante. Una fachada también puede simplificar portar sistemas a otras plataformas, ya que es menos probable que construir un subsistema requiera volver a construir todos los otros.

3. No impide que las aplicaciones usen las clases del subsistema en caso de que sea necesario. De este modo se puede elegir entre facilidad de uso y generalidad.

IMPLEMENTACIÓN

A la hora de implementar una fachada deben tenerse en cuenta los siguientes aspectos:

1. *Reducción del acoplamiento cliente-subsistema*. El acoplamiento entre clientes y el subsistema puede verse reducido todavía más haciendo que Fachada sea una clase abstracta con subclases concretas para las diferentes implementaciones de un subsistema. De esa manera los clientes pueden comunicarse con el subsistema a través de la interfaz de una clase abstracta Fachada. Este acoplamiento abstracto evita que los clientes tengan que saber qué implementación de un subsistema están usando.

   Una alternativa a la herencia es configurar un objeto Fachada con diferentes objetos del subsistema. Para personalizar la fachada basta con reemplazar uno o varios de tales objetos.

2. *Clases del subsistema públicas o privadas*. Un subsistema se parece a una clase en que ambos tienen interfaces y los dos encapsulan algo —una clase encapsula estado y operaciones, mientras que un subsistema encapsula clases —. Y del mismo modo que resulta útil pensar en la interfaz pública y privada de una clase, también podemos pensar

en la interfaz pública y privada de un subsistema.

La interfaz pública de un subsistema consiste en una serie de clases a las que acceden todos los clientes; la interfaz privada es sólo para quienes vayan a ampliar el subsistema. La clase Fachada es parte de la interfaz pública, por supuesto, pero no es la única. Otras clases del subsistema también suelen ser públicas. Por ejemplo, las clases Léxico y Sintáctico del subsistema de compilación son parte de la interfaz pública.

Sería interesante poder hacer privadas a las clases de un subsistema, pero hay pocos lenguajes orientados a objetos que lo permitan. Tanto C++ como Smalltalk han tenido tradicionalmente un espacio de nombres global para las clases. No obstante, recientemente, el comité de estandarización de C++ añadió espacios de nombres al lenguaje [Str94], lo que nos permitirá hacer visibles sólo las clases públicas del subsistema.

CÓDIGO DE EJEMPLO

Veamos más en detalle cómo introducir una fachada en un subsistema de compilación.

El subsistema de compilación define una clase FlujoDeCodigoBinario que implementa un flujo de objetos InstruccionBinaria. Un objeto InstruccionBinaria encapsula un código binario, que puede especificar las instrucciones en lenguaje máquina. El subsistema también define una clase Token para los objetos que encapsula tokens del lenguaje de programación.

La clase Lexico toma un flujo de caracteres y produce un flujo de tokens, un token cada vez.

```
class Lexico {
public:
     Lexico(istream&);
     virtual ~Lexico();

     virtual Token& Analizar();
private:
     istream& _flujoDeEntrada;
```

```
};
```

La clase Sintactico usa un ConstructorNodoPrograma para construir un árbol de análisis sintáctico a partir de los tokens del Lexico.

```
class Sintactico {
public:
     Sintactico();
     virtual ~Sintactico();

     virtual      void      Analizar(Lexico&,
ConstructorNodoPrograma&);
};
```

Sintactico realiza llamadas a ConstructorNodoPrograma para crear incrementalmente el árbol de análisis sintáctico. Estas clases interactúan según el patrón Builder (89).

```
class ConstructorNodoPrograma {
public:
     ConstructorNodoPrograma();

     virtual NodoPrograma* NuevaVariable(
          const char* nombreVariable
     ) const;

     virtual NodoPrograma* NuevaAsignacion(
          NodoPrograma*             variable,
NodoPrograma* expresion
     ) const;

     virtual                    NodoPrograma*
NuevaExpresionReturn(
          NodoPrograma* valor
     ) const;

     virtual NodoPrograma* NuevaCondicion(
          NodoPrograma* condicion,
          NodoPrograma*            parteTrue,
NodoPrograma* parteFalse
     ) const;
     // …
```

```
     NodoPrograma* ObtenerNodoRaiz();
private:
     NodoPrograma* _nodo;
};
```

El árbol de análisis se compone de instancias de subclases de NodoPrograma tales como NodoInstruccion, NodoExpresion y así sucesivamente. La jerarquía de NodoPrograma es un ejemplo de patrón Composite (151). NodoPrograma define una interfaz para manipular un nodo de programa y sus hijos, en caso de que existan.

```
class NodoPrograma {
public:
     // manipulación del nodo de programa
     virtual  void  ObtenerPosicionFuente(int&
linea, int& indice);
     // …

     // manipulación da los hijos
     virtual void Insertar(NodoPrograma*);
     virtual void Borrar(NodoPrograma*);
     // …

     virtual                              void
Recorrer(GeneradorDeCodigo&);
protected:
     NodoPrograma();
};
```

La operación Recorrer toma un objeto GeneradorDeCodigo. Las subclases de NodoPrograma usan este objeto para generar código máquina, empleando para ello objetos InstruccionBinaria sobre un FlujoDeCodigoBinario. La clase GeneradorDeCodigo es un visitante (*véase* el patrón Visitor (305)).

```
class GeneradorDeCodigo {
public;
     virtual void Visitar(NodoInstruccion*);
     virtual void Visitar(NodoExpresion*);
     // …
protected:
```

```
     GeneradorDeCodigo(FlujoDeCodigoBinario&);
protected:
     FlujoDeCodigoBinario& _salida;
};
```

GeneradorDeCodigo tiene subclases como GeneradorDeCodigoMaquinaDePila y GeneradorDeCodigoRISC, que generan código máquina para distintas arquitecturas hardware.

Cada subclase de NodoPrograma implementa Recorrer como una llamada a Recorrer sobre sus objetos hijo NodoPrograma. A su vez, cada hijo hace lo mismo para sus hijos, y así sucesivamente, de forma recursiva. Por ejemplo, NodoExpresion define Recorrer de la siguiente manera:

```
void                    NodoExpresion::Recorrer
(GeneradorDeCodigo& gc) {
     gc.Visitar(this);

     IteradorLista<NodoPrograma*> i(_hijos);

     for    (i.Primero();    !i.HaTerminado();
i.Siguiente()) {
          i.ElementoActual()->Recorrer(gc);
     }
}
```

Las clases descritas hasta ahora forman el subsistema de compilación. Ahora introduciremos una clase Compilador, una fachada que junta todas estas piezas. Compilador proporciona una interfaz simple para compilar código fuente y generar código para una determinada máquina.

```
class Compilador {
public:
     Compilador();

     virtual      void      Compilar(istream&,
FlujoDeCodigoBinario&);
};

void Compilador::Compilar (
```

```
     istream& entrada, FlujoDeCodigoBinario&
salida
) {
     Lexico lexico(entrada);
     ConstructorNodoPrograma constructor;
     Sintactico sintactico;

     sintactico.Analizar(lexico,
constructor);
     GeneradorDeCodigoRISC generador(salida);
     NodoPrograma*     arbolDeAnalisis     =
constructor.ObtenerNodoRaiz();
     arbolDeAnalisis->Recorrer(generador);
}
```

Esta implementation liga al código el tipo de generador de código a utilizar, de manera que los programadores no necesitan especificar la arquitectura de destino. Eso podría ser razonable si siempre fuera a haber una única arquitectura. Si ése no es el caso, tal vez queramos cambiar el constructor de Compilador para que tome un parámetro GaneradorDeCodigo. De ese modo los programadores pueden especificar el generador a usar cuando crean una instancia de Compilador. La fachada del compilador puede parametrizar otros participantes, como Lexico y ConstructorNodoPrograma, lo que añade flexibilidad, pero también se aparta de la misión del patrón Facade, que es simplificar la interfaz para el caso general.

USOS CONOCIDOS

El ejemplo del compilador de la sección Código de Ejemplo está inspirado en el sistema de compilación ObjectWorksVSmalltalk [Par90].

En el framework de aplicaciones ET++ [WGM88], una aplicación puede tener incorporadas herramientas de inspección de objetos en tiempo de ejecución. Estas herramientas están implementadas en un subsistema aparte que incluye una clase Fachada denominada “ProgrammingEnvironment”. Esta fachada define operaciones tales como InspectObject e InspectClass para acceder a las herramientas de inspección.

Una aplicación ET++ también puede prescindir de las capacidades de inspección proporcionadas. En ese caso,

ProgrammingEnvironment implementa estas peticiones como operaciones nulas; es decir, no hacen nada. Sólo la subclase ETProgrammingEnvironment implementa estas peticiones con operaciones que muestran los correspondientes inspectores. La aplicación no sabe si está o no disponible un entorno de inspección; hay un acoplamiento abstracto entre la aplicación y el subsistema de inspección de objetos.

El sistema operativo Choices [CIRM93] usa fachadas para combinar varios frameworks en uno solo. Las abstracciones clave en Choices son los procesos, el almacenamiento y los espacios de direcciones. Para cada una de estas abstracciones existe su correspondiente subsistema, implementado como un framework, lo que permite portar Choices a diversas plataformas hardware. Dos de estos subsistemas tienen un “representante” (es decir, una fachada). Estos representantes son FileSystemInterface (para el almacenamiento) y Domain (para los espacios de direcciones).

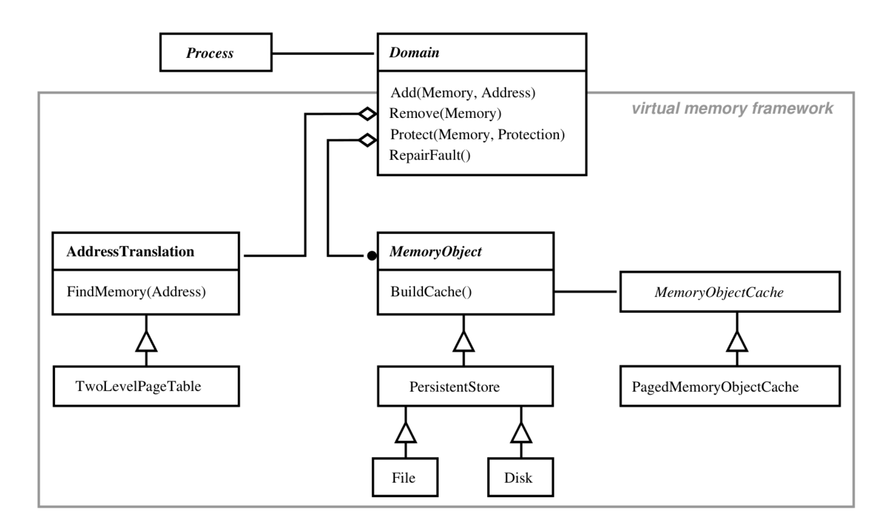
Por ejemplo, el framework de memoria virtual tiene como fachada a la clase Domain (dominio). Un dominio representa un espacio de direcciones, y proporciona una correspondencia entre las direcciones virtuales y los desplazamientos, por un lado, y los objetos en la memoria, ficheros y copias de seguridad, por otro. Las operaciones principales de Domain permiten añadir un objeto en la memoria a una determinada dirección, eliminar un objeto de la memoria y procesar los fallos de página.

Como se muestra en el siguiente diagrama, el subsistema de memoria virtual usa internamente los siguientes componentes:

- MemoryObject representa un almacenamiento de datos.
- MemoryObjectCache guarda en la memoria física los datos de varios MemoryObject. MemoryObjectCache es realmente una Estrategia (289) que localiza la política de caché.
- AddressTranslation encapsula el hardware de traducción de direcciones.

Cada vez que tiene lugar una interrupción por fallo de página se llama a la operación RepairFault. El objeto Domain busca el objeto de memoria de la dirección causante del fallo de página y delega la operación RepairFault en la caché asociada a ese objeto de memoria. Los dominios pueden personalizarse cambiando sus

componentes.



PATRONES RELACIONADOS

El patrón Abstract Factory (79) puede usarse para proporcionar una interfaz para crear el subsistema de objetos de forma independiente a otros subsistemas. Las fábricas abstractas también pueden ser una alternativa a las fachadas para ocultar clases especificas de la plataforma.

El patrón Mediator (251) es parecido al Facade en el sentido de que abstrae funcionalidad a partir de unas clases existentes. Sin embargo, el propósito del Mediador es abstraer cualquier comunicación entre objetos similares, a menudo centralizando la funcionalidad que no pertenece a ninguno de ellos. Los colegas de un mediador sólo se preocupan de comunicarse con él y no entre ellos directamente. Por el contrario, una fachada simplemente abstrae una interfaz para los objetos del subsistema, haciéndolos más fácil de usar; no define nueva funcionalidad, y las clases del subsistema no saben de su existencia.

Normalmente sólo necesita un objeto Fachada. Por tanto, éstos suelen implementarse como Singletons (119).

Estructural de Objetos

FLYWEIGHT
(Peso Ligero)

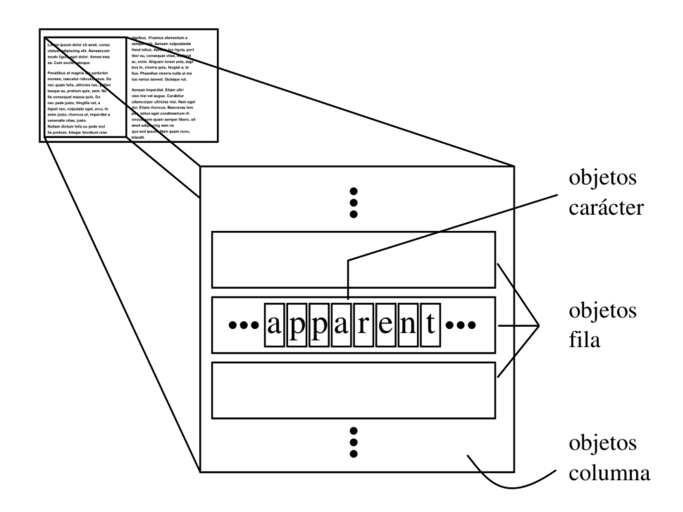
PROPÓSITO

Usa compartimiento para permitir un gran número de objetos de grano fino de forma eficiente.

MOTIVACIÓN

Si bien algunas aplicaciones podrían beneficiarse de un diseño en el que se empleasen objetos para todo, una implementación simplista haría que éste fuese prohibitivamente caro.

Por ejemplo, la mayoría de las implementaciones de editores de documentos tienen funciones de formateado y edición de texto que son hasta cierto punto modulares. Los editores de documentos orientados a objetos normalmente emplean objetos para representar los elementos insertados, como tablas y figuras. Sin embargo, no suelen hacer uso de un objeto para cada uno de los caracteres del documento, a pesar de que de esa manera se lograría una gran flexibilidad en la aplicación. En ese caso se podría tratar de manera uniforme a los caracteres y a los elementos insertados con respecto a cómo se dibujan y formatean éstos. La aplicación podría ampliarse para permitir nuevos juegos de caracteres sin afectar al resto de su funcionalidad. La estructura de objetos de la aplicación podría mimetizar la estructura física del documento. El siguiente diagrama muestra cómo puede usar objetos un editor de documentos para representar los caracteres.



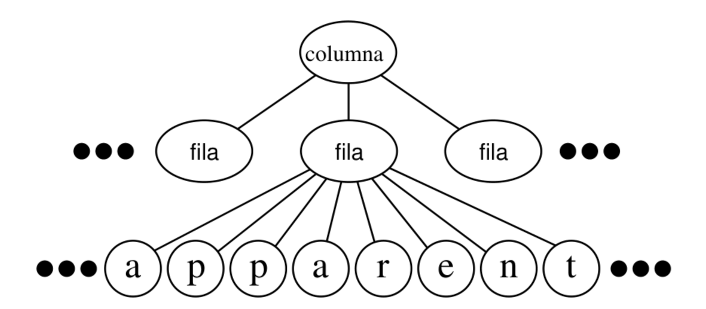
El inconveniente de este diseño es su coste. Incluso documentos de un tamaño moderado podrían necesitar cientos o miles de objetos de caracteres, los cuales consumirían mucha memoria y podrían sufrir un coste en tiempo de ejecución inaceptable. El patrón Flyweight describe cómo compartir objetos para permitir su uso con granularidades muy finas sin un coste prohibitivo.

Un **peso ligero** es un objeto compartido que puede usarse a la vez en varios contextos. El peso ligero es un objeto independiente en cada contexto —no se puede distinguir de una instancia del objeto que no esté compartida—. Los pesos ligeros no pueden hacer suposiciones sobre el contexto en el cual operan. Lo fundamental aquí es la distinción entre estado intrínseco y extrínseco. El estado intrínseco se guarda con el propio objeto; consiste en información que es independiente de su contexto y que puede ser, por tanto, compartida. El estado extrínseco depende del contexto y cambia con él, por lo que no puede ser compartido. Los objetos cliente son responsables de pasar al peso ligero su estado extrínseco cuando lo necesite.

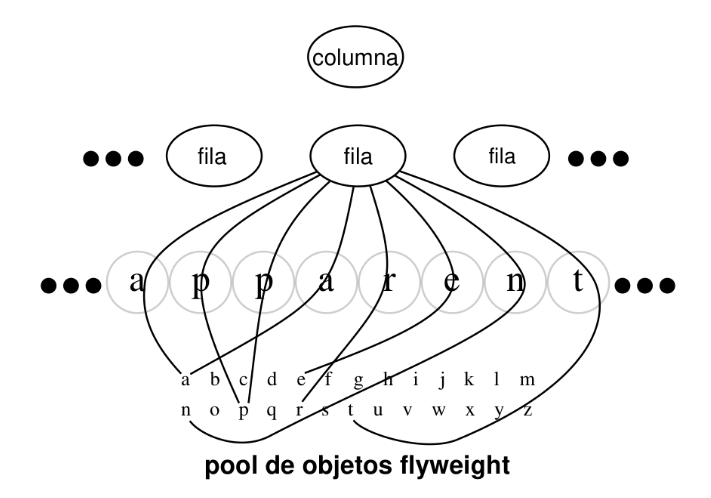
Los pesos ligeros modelan conceptos o entidades que normalmente son demasiado numerosos como para ser representados con objetos. Por ejemplo, un editor de documentos puede crear un peso ligero para cada letra del alfabeto. Cada peso ligero guarda un código de carácter, mientras que las coordenadas con su posición en el documento y su estilo tipográfico pueden

obtenerse a partir de los algoritmos de maquetación del texto y las órdenes de formateado activas allí donde aparezca dicho carácter. El código del carácter es su estado intrínseco, mientras que el resto de la información es extrínseca.

Desde un punto de vista lógico, hay un objeto por cada aparición en el documento de un determinado carácter:

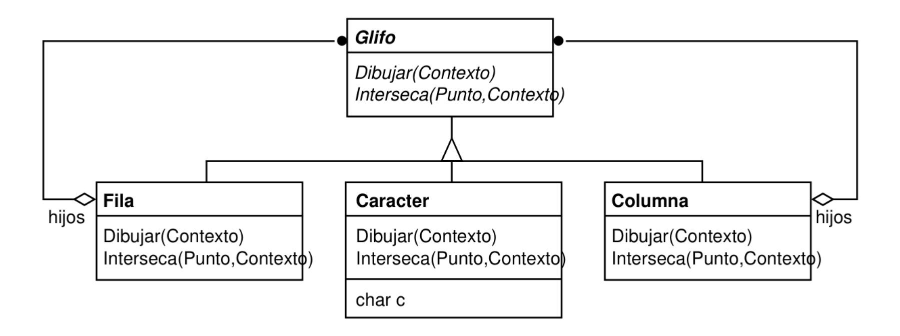


Físicamente, no obstante, hay un solo objeto peso ligero compartido por cada carácter, el cual aparece en diferentes contextos de la estructura del documento. Cada aparición de un carácter concreto se refiere a la misma instancia del almacén[39] compartido de objetos peso ligero:



A continuación se muestra la estructura de clases de estos objetos. Glifo es la clase abstracta para los objetos gráficos, algunos de los cuales pueden ser pesos ligeros. Las operaciones que pueden

depender del estado extrínseco reciben éste como parámetro. Por ejemplo, Dibujar e Interseca deben saber en qué contexto está el glifo antes de que puedan hacer su trabajo.



Un peso ligero que represente la letra "a" únicamente guarda el código de carácter correspondiente; no necesita guardar su posición ni tipo de fuente. Los clientes proporcionan la información dependiente del contexto que el peso ligero necesita para dibujarse a s(mismo. Por ejemplo, una Fila sabe dónde deberían dibujarse sus hijos para que se dispongan horizontalmente. Por tanto, puede pasar a cada hijo su posición cuando les pida que se dibujen.

Puesto que el número de objetos de carácter diferentes es bastante menor que el número de caracteres del documento, el número total de objetos es notablemente menor que los que usaría una implementación simplista. Un documento en el que todos los caracteres aparezcan con la misma fuente y color tendrá del orden de 100 objetos de carácter (aproximadamente el tamaño del juego de caracteres ASCII), independientemente de cuál sea su longitud. Por otro lado, dado que la mayoría de los documentos no usan más de 10 combinaciones diferentes de fuentes y colores, este número no crecerá demasiado en la práctica. Emplear un objeto como abstracción para los caracteres individuales es, por tanto, práctico.

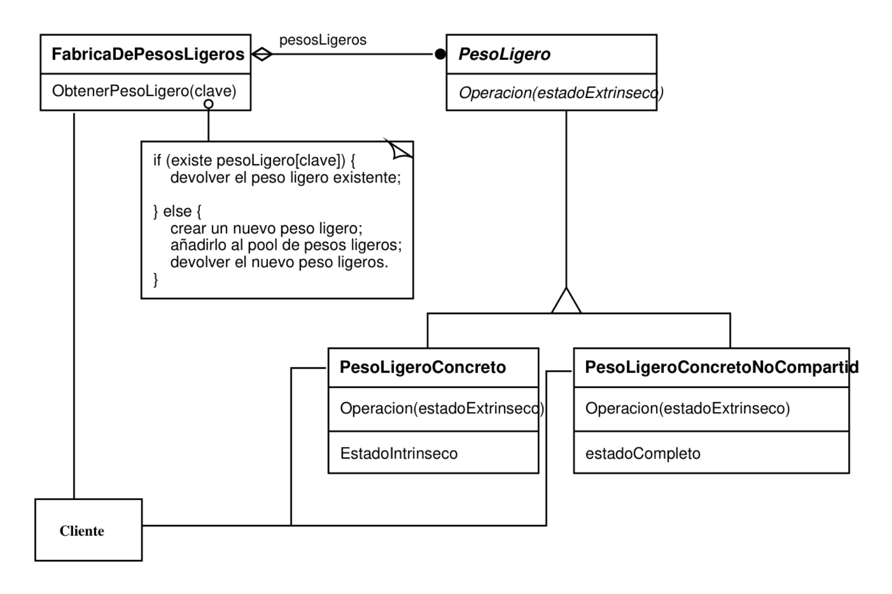
APLICABILIDAD

La efectividad del patrón Flyweight depende enormemente de cómo y dónde se use. Debería aplicarse el patrón cuando se cumpla *todo* lo siguiente:

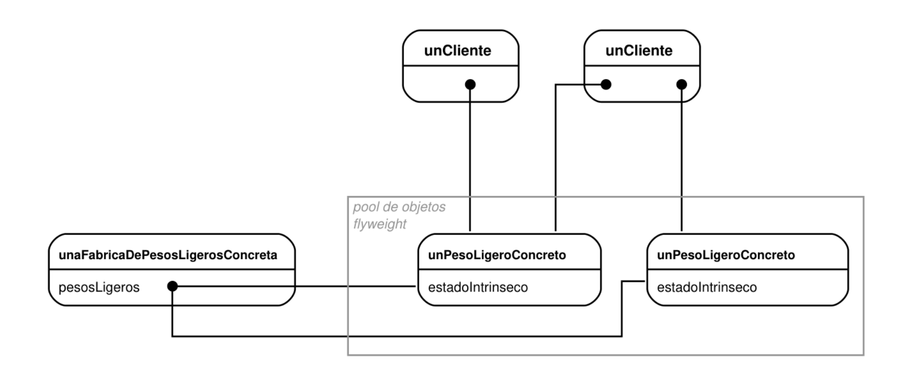
- Una aplicación utiliza un gran número de objetos.

- Los costes de almacenamiento son elevados debido a la gran cantidad de objetos.
- La mayor parte del estado del objeto puede hacerse extrínseco.
- Muchos grupos de objetos pueden reemplazarse por relativamente pocos objetos compartidos, una vez que se ha eliminado el estado extrínseco.
- La aplicación no depende de la identidad de un objeto. Puesto que los objetos peso ligero pueden ser compartidos, las comprobaciones de identidad devolverán verdadero para objetos conceptualmente distintos.

ESTRUCTURA



El siguiente diagrama de objetos muestra cómo se comparten los pesos ligeros:



PARTICIPANTES

- **PesoLigero** (Glifo)
  - declara una interfaz a través de la cual los pesos ligeros pueden recibir un estado extrínseco y actuar sobre él.

El patrón Flyweight suele combinarse con el patrón Composite para representar una estructura jerárquica como un grafo con nodos hojas compartidos. Una consecuencia del compartimiento es que los nodos hojas pesos ligeros no pueden tener un puntero a su padre. En vez de eso, el puntero al padre se le pasa al peso ligero como parte de su estado extrínseco. Esto tiene un impacto decisivo en la forma en que se comunican los objetos de la jerarquía.

IMPLEMENTACIÓN

A la hora de implementar el patrón Peso Ligero han de tenerse presentes las siguientes cuestiones:

1. *Eliminar el estado extrínseco*. La aplicabilidad del patrón viene dada en gran medida por lo fácil que sea identificar el estado extrínseco y eliminarlo de los objetos compartidos. Eliminar el estado extrínseco no ayudará a reducir los costes de almacenamiento en caso de que haya tantos tipos diferentes de estados extrínsecos como objetos hubiera antes del compartimiento. Lo ideal sería que el estado extrínseco pudiera ser calculado a partir de una estructura de objetos separada que tuviera requisitos de

almacenamiento mucho menores.

En nuestro editor de documentos, por ejemplo, podemos almacenar un diccionario con información tipográfica en una estructura aparte en vez de almacenar la fuente y el estilo con cada objeto de carácter. Dicha estructura de datos almacena series de caracteres con los mismos atributos tipográficos. Cuando un carácter se dibuja a sí mismo recibe sus atributos tipográficos como un efecto lateral de la operación de dibujado. Como los documentos normalmente sólo hacen uso de unas pocas fuentes y estilos diferentes, almacenar esta información fuera de cada objeto de carácter es mucho más eficiente que si se almacenase internamente.

2. *Gestionar los objetos compartidos*. Dado que los objetos están compartidos, los clientes no deberían crear instancias de ellos directamente. La FabricaDePesosLigeros permite que los clientes obtengan un peso ligero concreto. Para ello, suelen utilizar un almacén asociativo donde los clientes pueden buscar los objetos pesos ligeros de su interés. Por ejemplo, la fábrica de pesos ligeros del ejemplo del editor de documentos podría mantener una tabla con objetos pesos ligeros indexados por códigos de carácter. El administrador devuelve el objeto peso ligero apropiado para un código dado, creando el objeto peso ligero en caso de que éste no existiese.

   El compartimiento también implica alguna forma de conteo de referencias o de recolección de basura para recuperar el espacio de almacenamiento de un peso ligero cuando éste ya no sea necesario. Sin embargo, ninguna de estas cosas es necesaria si hay un número fijo y pequeño de tales objetos (por ejemplo, los pesos ligeros para el juego de caracteres ASCII). En ese caso, vale la pena mantener los pesos ligeros de forma permanente.

CÓDIGO DE EJEMPLO

Volviendo a nuestro ejemplo del formateador de documentos,

podemos definir una clase base Glifo base para los objetos gráficos. Como es lógico, los glifos son Compuestos (*véase* el patrón Composite (151)) que tienen atributos gráficos y que pueden dibujarse a sí mismos. Aquí nos centraremos simplemente en el atributo para la fuente, pero podría seguirse el mismo enfoque para cualquiera de los otros atributos gráficos que pudiera tener un glifo.

```
class Glifo {
public:
     virtual ~Glifo();

     virtual      void      Dibujar(Ventana*,
ContextoGlifo&);
     virtual  void  EstablecerFuente(Fuente*,
ContextoGlifo&);
     virtual                         Fuente*
ObtenerFuente(ContextoGlifo&);

     virtual void Primero(ContextoGlifo&);
     virtual void Siguiente(ContextoGlifo&);
     virtual                            bool
HaTerminado(ContextoGlifo&);
     virtual Glifo* Actual(ContextoGlifo&);

     virtual      void      Insertar(Glifo*,
ContextoGlifo&);
     virtual void Borrar(ContextoGlifo&);
protected:
     Glifo();
};
```

La subclase Caracter simplemente almacena un código de carácter:

```
clase Caracter : public Glifo {
public:
     Caracter(char);

     virtual      void      Dibujar(Ventana*,
ContextoGlifo&);
private:
     char _codigocaracter;
);
```

Para evitar reservar espacio en cada glifo para el atributo de la fuente, almacenaremos dicho atributo extrínsecamente en un objeto ContextoGlifo, el cual sirve de repositorio para el estado extrínseco, manteniendo una correspondencia entre un glifo y su fuente (y otros atributos gráficos cualesquiera que pudiera tener) en diferentes contextos. Cualquier operación que necesite saber la fuente del glifo en un contexto dado recibirá como parámetro una instancia de un objeto ContextoGlifo. La operación puede entonces pedirle a ContextoGlifo cuál es la fuente en ese contexto. El contexto depende de la posición del glifo en la estructura. Por tanto, las operaciones de iteración y manipulación sobre los hijos del Glifo deben actualizar el ContextoGlifo cada vez que se usan.

```
class ContextoGlifo {
public:
     ContextoGlifo();
     virtual ~ContextoGlifo();

     virtual  void  Siguiente(int  incremento
=1);
     virtual void Insertar(int cantidad = 1);

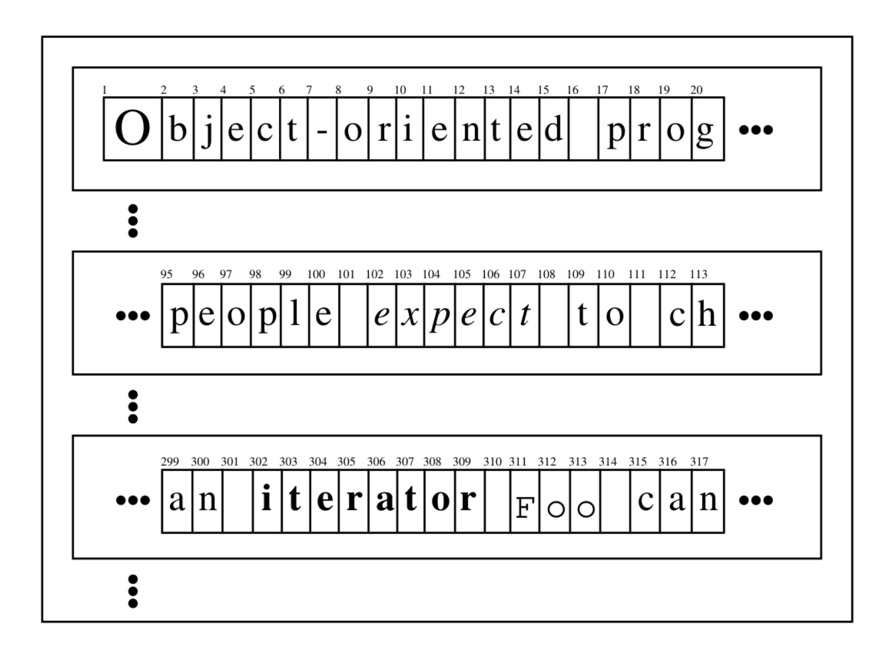
     virtual Fuente* ObtenerFuente();
     virtual  void  EstablecerFuente(Fuente*,
int span = 1);
private:
     int _indice;
     ArbolB* _fuentes;
};
```

El ContextoGlifo debe permanecer informado de la posición actual en la estructura de glifos durante el recorrido. ContextoGlifo::Siguiente incrementa _indice a medida que se avanza en el recorrido. Las subclases de Glifo que tienen hijos (como Fila y Columna) deben implementar Siguiente para que llame a ContextoGlifo::Siguiente en cada punto del recorrido.

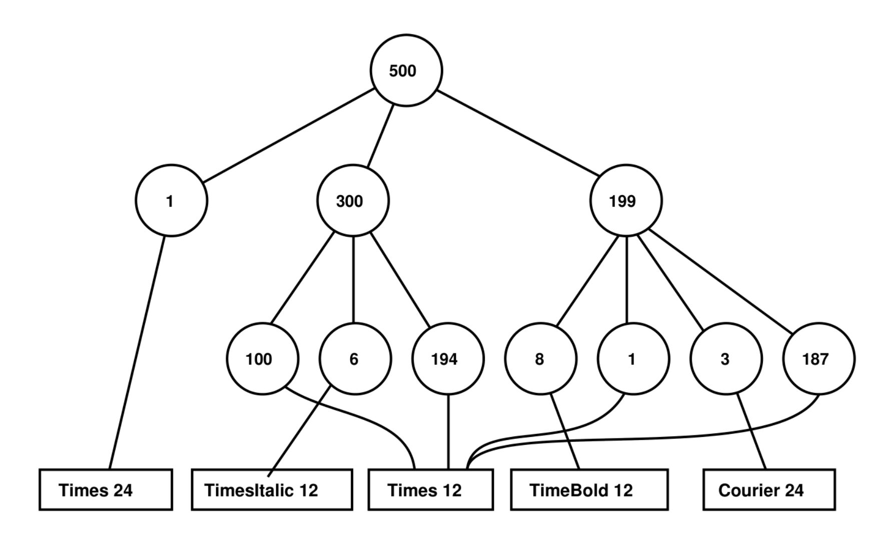
ContextoGlifo::ObtenerFuente usa el índice como clave de una estructura ArbolB que guarda la correspondencia entre glifos y fuentes. Cada nodo del árbol se etiqueta con la longitud de la cadena para la que se proporciona información sobre la fuente. Las

hojas del árbol apuntan a una fuente, mientras que los nodos interiores separan la cadena en subcadenas, una para cada hijo.

Sea el siguiente extracto de una composición de glifos:



La estructura ArbolB para la información sobre la fuente podría parecerse a esto:

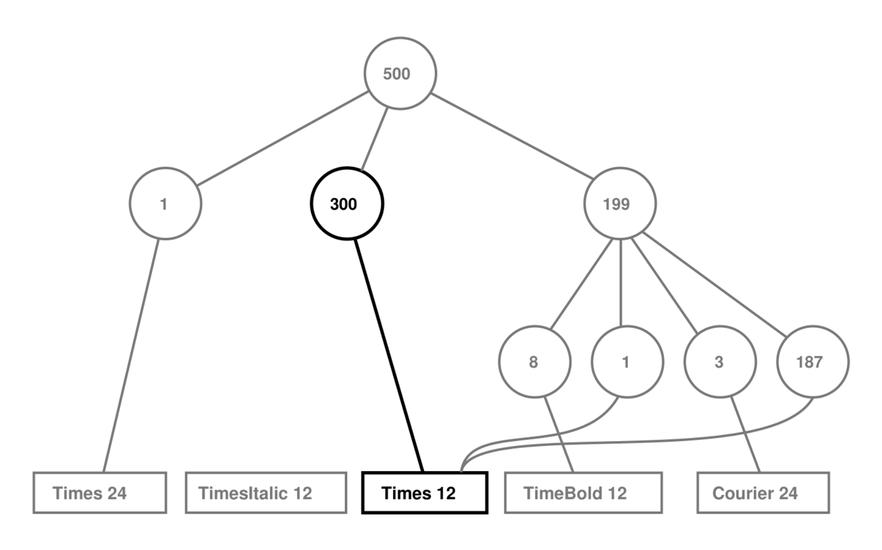


Los nodos interiores definen intervalos de índices de glifos. El ArbolB se actualiza en respuesta a los cambios de fuentes y cada vez que se añaden o se eliminan glifos de la estructura. Por ejemplo, suponiendo que nos encontramos en el índice 102 durante un recorrido, el siguiente código establece la fuente de cada carácter de la palabra "expect" como la misma que la del texto adyacente (es decir, times12, una instancia de Fuente para Times Roman de 12 puntos):

```
ContextoGlifo contexto;
Fuente*   times12   =   new   Fuente("Times-
Roman-12");
Fuente*  timesItalic12  =  new  Fuente("Times-
Italic-12");
// …

gc.EstablecerFuente(times12, 6);
```

La nueva estructura ArbolB (con los cambios resaltados en negro) sé parecería a



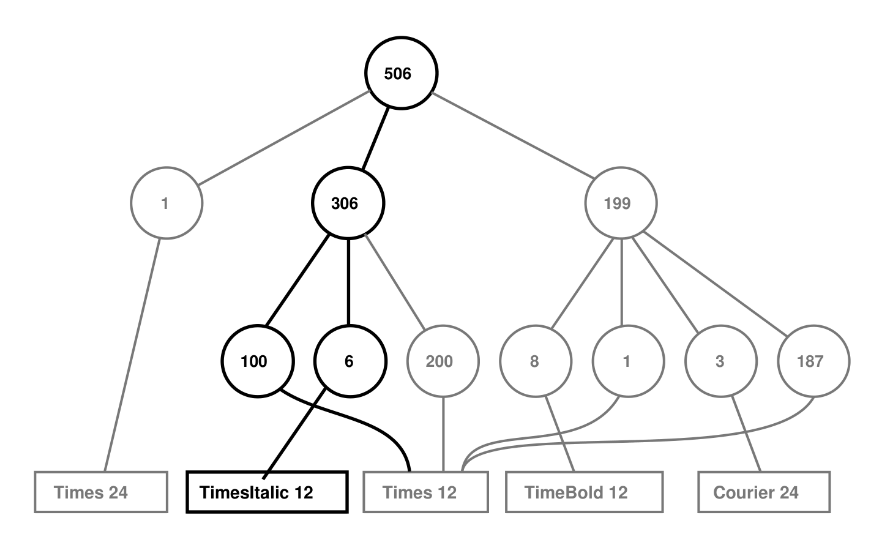
Supongamos que añadimos la palabra
"don't"
(incluyendo un espacio en blanco) en Times Italic de 12 puntos

antes de “expect”. El código siguiente informa al contexto de este evento, suponiendo que todavía se encuentre en el índice 102:

```
contexto.Insertar(6);
contexto.EstablecerFuente(timesItalic12, 6);
```

La estructura ArbolB se convierte en lo que aparece en la figura de la página siguiente.

Cuando se le pide a ContextoGlifo la fuente del glifo actual, éste desciende por el ArbolB, añadiendo índices hasta que encuentra la fuente para el índice actual. Dado que la frecuencia de los cambios de fuente es relativamente baja, el árbol se mantiene pequeño en comparación con el tamaño de la estructura de glifos. Esto mantiene bajos los costes de almacenamiento sin un aumento desorbitado del tiempo de búsqueda **[40]** .



El último objeto que necesitamos es una FabricaDePesosLigeros que cree glifos y garantice que se comparten de manera adecuada. La clase FabricaDeGlifos crea instancias de Carácter y otros tipos de glifos. Nosotros sólo compartimos objetos Carácter; los glifos compuestos son más raros y, en cualquier caso, su estado importante (es decir, sus hijos) es intrínseco.

```
const int NCODIGOSCARACTER = 128;

class FabricaDeGlifos {
public:
     FabricaDeGlifos();
     virtual ~FabricaDeGlifos();

     virtual Caracter* CrearCaracter(char);
     virtual Fila* CrearFila();
     virtual Columna* CrearColumna();
     // …
private:
     Carácter* _caracter[NCODIGOSCARACTER];
};
```

El array _caracter contiene punteros a glifos Carácter indexados por el código de carácter. Este array es inicializado a cero en el constructor.

```
FabricaDeGlifos::FabricaDeGlifos () {
     for (int i = 0; i < NCODIGOSCARACTER; +
+i) {
          _caracter[i] = 0;
     }
}
```

CrearCaracter busca un carácter en los glifos de carácter del array, y devuelve el glifo correspondiente, si existe. Si no existe, entonces CrearCaracter crea el glifo, lo mete en el array y lo devuelve:

```
Caracter*     FabricaDeGlifos::CrearCaracter.
(char c) {
     if (!_caracter[c]) {
          _caracter[c] = new Caracter(c);
     }

     return _caracter[c];
}
```

El resto de operaciones simplemente crean un nuevo objeto cada vez que son llamadas, dado que los glifos que no sean caracteres no

serán compartidos:

```
Fila* FabricaDeGlifos::CrearFila () {
     return new Fila;
}

Columna* FabricaDeGlifos::CrearColumna () {
     return new Columna;
}
```

Podríamos omitir estas operaciones y dejar que los clientes crearan instancias de glifos no compartidos directamente. Sin embargo, en caso de que más tarde decidiésemos hacer dichos glifos compartidos, tendríamos que cambiar el código cliente que los crea.

USOS CONOCIDOS

El concepto de objetos pesos ligeros se describió e investigó por vez primera como una técnica de diseño en Interviews 3.0 [CL90]. Sus desarrolladores construyeron un potente editor de documentos llamado Doc para demostrar el concepto [CL92]. Doc usa objetos glifo para representar cada carácter del documento. El editor crea una instancia de Glifo para cada carácter que tenga un determinado estilo (el cual define sus atributos gráficos); por tanto el estado intrínseco de un carácter consiste en el código del carácter y su información de estilo (un índice en la tabla de estilos)[41]. Eso significa que únicamente la posición es extrínseca, lo que hace que Doc sea rápido. Los documentos se representan por una clase Document, que también hace las veces de FabricaDePesosLigeros. Las mediciones hechas sobre Doc muestran que el compartimiento de caracteres resulta efectivo. Un documento típico de
180 000
caracteres requirió solamente 480 objetos de carácter.

ET++ [WGM88] usa pesos ligeros para permitir la independencia de la interfaz de usuario[42]. El estándar de interfaz de usuario[43] afecta a la disposición de los elementos de la interfaz de usuario (por ejemplo, barras de desplazamiento, botones y menús —lo que se conoce con el nombre colectivo de “útiles”—)

[44] y sus adornos (sombras, bordes, etc.). Un útil delega todo su comportamiento de posicionamiento y dibujado en otro objeto Layout. Cambiar el objeto Layout cambia el aspecto de la interfaz, incluso en tiempo de ejecución.

Para cada clase de útil hay su correspondiente clase Layout (por ejemplo, ScrollbarLayout, MenubarLayout, etc.). Un problema evidente con este enfoque es que usar diferentes objetos Layout duplica el número de objetos de interfaz de usuario: para cada objeto de interfaz de usuario hay un objeto Layout adicional. Para evitar esta penalización, los objetos Layout se implementan como pesos ligeros. Dichos objetos son buenos pesos ligeros porque su principal misión es definir comportamiento, y es fácil pasarles el poco estado extrínseco que necesitan para colocar o dibujar un objeto.

Los objetos Layout son creados y gestionados por objetos Look. La clase Look es una Fabrica Abstracta (79) que recupera un determinado objeto Layout con operaciones como GetButtonLayout, GetMenuBarLayout y otras. Para cada estándar de interfaz de usuario existe la correspondiente subclase de Look (como MotifLook u OpenLook) que proporciona los objetos Layout apropiados.

Por cierto, los objetos Layout son, en esencia, estrategias, (véase el patrón Strategy (289)). Son un ejemplo de objeto estrategia implementado como un peso ligero.

PATRONES RELACIONADOS

El patrón Flyweight suele combinarse con el patrón Composite (151) para implementar una estructura lógica jerárquica en términos de un grafo dirigido acíclico con nodos hojas compartidos.

Suele ser mejor implementar los objetos State (279) y Strategy (289) como pesos ligeros.

PROXY (Apoderado)

PROPÓSITO

Proporciona un representante o sustituto de otro objeto para controlar el acceso a éste.

TAMBIÉN CONOCIDO COMO

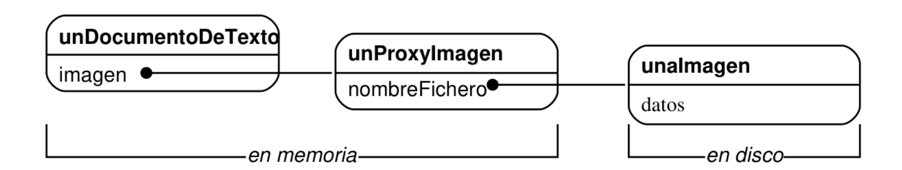
Surrogate (Sustituto)

MOTIVACIÓN

Una razón para controlar el acceso a un objeto es retrasar todo el coste de su creación e inicialización hasta que sea realmente necesario usarlo. Pensemos en un editor de documentos que puede insertar objetos gráficos en un documento. Algunos de estos objetos gráficos, como las grandes imágenes *raster*, pueden ser costosos de crear. Sin embargo, abrir un documento debería ser una operación que se efectuase rápidamente, por lo que se debería evitar crear todos los objetos costosos a la vez en cuanto se abre el documento. Por otro lado, tampoco es necesario, ya que no todos esos objetos serán visibles en el documento al mismo tiempo.

Estas restricciones sugieren que cada objeto concreto se cree *a petición*, lo que en este caso tendrá lugar cuando la imagen se hace visible. Pero ¿qué ponemos en el documento en lugar de la imagen? ¿Y cómo puede ocultarse el hecho de que la imagen se crea a petición sin que se complique la implementación del editor? Esta optimización no debería influir, por ejemplo, en el código de visualización y formateado.

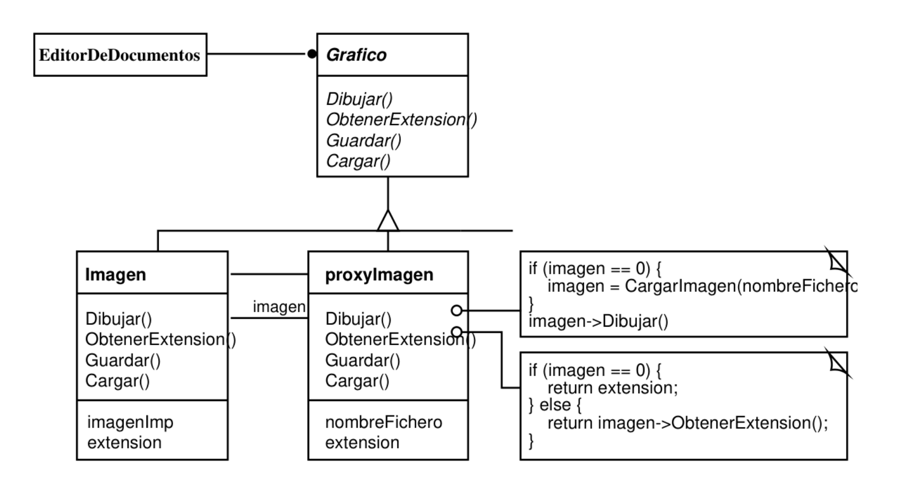
La solución es utilizar otro objeto, un **proxy** de la imagen, que actúe como un sustituto de la imagen real. El proxy se comporta igual que la imagen y se encarga de crearla cuando sea necesario.



El proxy de la imagen crea la imagen real sólo cuando el editor de documentos le pide que se dibuje, invocando a su operación Dibujar. El proxy redirige las siguientes peticiones directamente a la imagen. Debe guardar, por tanto, una referencia a la imagen después de crear ésta.

Supongamos que las imágenes se guardan en ficheros aparte. En este caso podemos usar el nombre del fichero como la referencia al objeto real. El proxy también almacena su **extensión**, esto es, su ancho y su alto. La extensión permite que el proxy pueda responder a las preguntas sobre su tamaño que le haga el formateador sin crear realmente la imagen.

El siguiente diagrama de clases ilustra este ejemplo más en detalle.



El editor de documentos accede a las Imágenes insertadas a través de la interfaz definida por la clase abstracta Gráfico. ProxyImagen es una clase para las imágenes que se crean a petición. ProxyImagen tiene como referencia a la imagen en el disco el nombre del fichero, el cual recibe como parámetro el constructor de la clase.

ProxyImagen también guarda la caja que circunscribe la Imagen

y una referencia a la instancia de la Imagen real, la cual no será válida hasta que el proxy cree dicha imagen real. La operación Dibujar se asegura de que la imagen ha sido creada antes de enviarle la petición. ObtenerExtension redirige la petición a la imagen sólo si ésta ha sido creada; en caso contrario, es ProxyImagen quien devuelve su extensión.

APLICABILIDAD

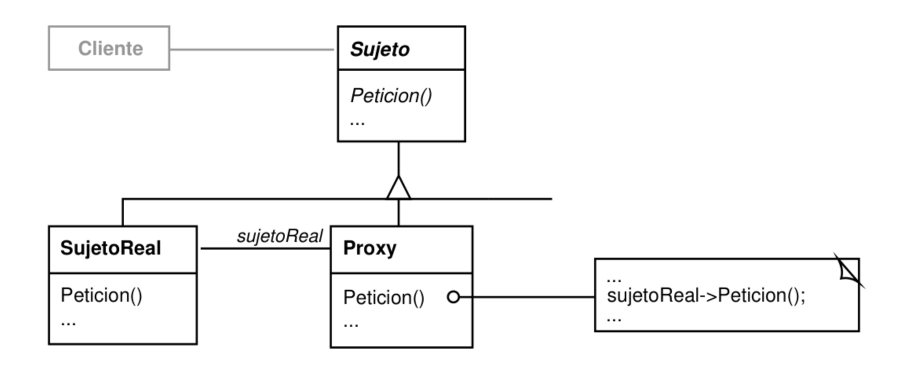
Este patrón es aplicable cada vez que hay necesidad de una referencia a un objeto más versátil o sofisticada que un simple puntero. Éstas son varias situaciones comunes en las que es aplicable el patrón Proxy:

1. Un **proxy remoto** proporciona un representante local de un objeto situado en otro espacio de direcciones. NEXTSTEP [Add94] usa la clase NXProxy con este propósito. Coplien [Cop92] llama "Embajador" a este tipo de proxy.
2. Un **proxy virtual** crea objetos costosos por encargo. El ProxyImagen descrito en la sección de Motivación es un ejemplo de este tipo de proxy.
3. Un **proxy de protección** controla el acceso al objeto original. Los proxies de protección son útiles cuando los objetos debieran tener diferentes permisos de acceso. Por ejemplo, los KernelProxy del sistema operativo Choices [CIRM93] proporcionan un acceso protegido a los objetos del sistema operativo.
4. Una **referencia inteligente** es un sustituto de un simple puntero que lleva a cabo operaciones adicionales cuando se accede a un objeto. Algunos ejemplos de usos típicos son:

    - contar el número de referencias al objeto real, de manera que éste pueda liberarse automáticamente cuando no haya ninguna referencia apuntándole (también se conocen con el nombre de **punteros inteligentes** [Ede92]).
    - cargar un objeto persistente en la memoria cuando es referenciado por primera vez.
    - comprobar que se bloquea el objeto real antes de

acceder a él para garantizar que no pueda ser modificado por ningún otro objeto.

ESTRUCTURA



Éste es un posible diagrama de objetos de una estructura de proxies en tiempo de ejecución:



PARTICIPANTES

- **Proxy** (ProxyImagen)
    - mantiene una referencia que permite al proxy acceder al objeto real. HI proxy puede referirse a un Sujeto en caso de que las interfaces de SujetoReal y Sujeto sean la misma.
    - proporciona una interfaz idéntica a la de Sujeto, de manera que un proxy pueda ser sustituido por el sujeto real.
    - controla el acceso al sujeto real, y puede ser responsable de su creación y borrado.
    - otras responsabilidades dependen del tipo de proxy:

    - los *proxies remotos* son responsables de codificar una petición y sus argumentos para enviar la petición codificada al sujeto real que se encuentra en un espacio de direcciones diferente.
    - los *proxies virtuales* pueden guardar información adicional sobre el sujeto real, por lo que pueden retardar el acceso al mismo. Por ejemplo, el ProxyImagen de la sección de Motivación guinda la extensión de la imagen real.
    - los *proxies de protección* comprueban que el llamador tenga los permisos de acceso necesarios para realizar una petición.
- **Sujeto** (Gráfico)
  - define la interfaz común para el SujetoReal y el Proxy, de modo que pueda usarse un Proxy en cualquier sitio en el que se espere un SujetoReal.
- **SujetoReal** (Imagen)
  - define el objeto real representado.

COLABORACIONES

El Proxy redirige peticiones al SujetoReal cuando sea necesario, dependiendo del tipo de proxy.

CONSECUENCIAS

El patrón Proxy introduce un nivel de indirección al acceder a un objeto. Esta indirección adicional tiene muchos posibles usos, dependiendo del tipo de proxy:

1. Un proxy remoto puede ocultar el hecho de que un objeto reside en un espacio de direcciones diferente.
2. Un proxy virtual puede llevar a cabo optimizaciones tales como crear un objeto por encargo.
3. Tanto los proxies de protección, como las referencias inteligentes, permiten realizar tareas de mantenimiento adicionales cuando se accede a un objeto.

Hay otra optimización que el patrón Proxy puede ocultar al cliente. Se denomina copia-de-escritura, y está relacionada con la creación por encargo. Copiar un objeto grande y complejo puede ser una operación costosa. Si dicha copia no se modifica nunca, no hay necesidad de incurrir en ese gasto. Al utilizar un proxy para posponer el proceso de copia nos estamos asegurando de que sólo pagamos el precio de copiar el objeto en caso de que éste sea modificado.

Para realizar una copia-de-escritura el sujeto debe tener un contador de referencias. Copiar el proxy no será más que incrementar dicho contador. Sólo cuando el cliente solicita una operación que modifica el sujeto es cuando el proxy realmente lo copia. En ese caso el proxy también tiene que disminuir el contador de referencias del sujeto. Cuando éste llega a cero se borra el sujeto.

La copia-de-escritura puede reducir significativamente el coste de copiar sujetos pesados.

IMPLEMENTACIÓN

El patrón Proxy puede explotar las siguientes características de los lenguajes:

1. *Sobrecargar el operador de acceso a miembros en* C++. C++ admite la sobrecarga de operator->, el operador de acceso a miembros. Sobrecargar este operador permite realizar tareas adicionales cada vez que se desreferencia un objeto, lo que puede resultar útil para implementar algunos tipos de proxies; el proxy se comporta igual que un puntero.

   El siguiente ejemplo ilustra cómo usar esta técnica para implementar un proxy virtual llamado PunteroImagen.

```
class Imagen;
extern                          Imagen*
CargarUnFicheroDeImagen(const
char*);
     // función externa
class PunteroImagen {
public:
```

```
     PunteroImagen(const       char*
FicheroDeImagen);
     virtual ~PunteroImagen();

     virtual Imagen* operator->();
     virtual Imagen& operator*();
private:
     Imagen* CargarImagen();
private:
     Imagen* _imagen;
     const char* _ficheroDeImagen;
};

PunteroImagen::PunteroImagen   (const
char* elFicheroDeImagen) {
     _ficheroDeImagen                =
elFicheroDeImagen;
     imagen = 0;
}

Imagen*   PunteroImagen::CargarImagen
() {
     if (_imagen == 0) {
          _imagen                    =
CargarUnFicheroDeImagen(_ficheroDeImagen);
     }
     return imagen;
}
```

Los operadores sobrecargados -> y * usan CargarImagen para devolver _imagen a sus llamadores (cargando la imagen si es necesario).

```
Imagen* PunteroImagen::operator-> ()
{
     return CargarImagen();
}

Imagen&  PunteroImagen::operator*  ()
{
     return *CargarImagen();
}
```

Este enfoque permite llamar a las operaciones de Imagen a

través de objetos PunteroImagen sin el problema de hacer a esas operaciones parte de la interfaz de PunteroImagen:

```
PunteroImagen         imagen         =
PunteroImagen("nombreDeUnFicheroDeImagen");
imagen->Dibujar(Punto(50, 100));
     //        (Imagen.operator->())-
>Dibujar(Punto(50, 100))
```

Nótese cómo el proxy de imagen funciona como un puntero, pero no está declarado como un puntero a Imagen. Eso significa que no se puede usar exactamente igual que un puntero real a Imagen. Por tanto, con este enfoque los clientes deben tratar de forma diferente a los objetos Imagen y PunteroImagen.

Sobrecargar el operador de acceso a miembros no es una buena solución para todos los tipos de proxies. Algunos necesitan saber exactamente *qué* operación es llamada, y en esos casos no sirve sobrecargar el operador de acceso a miembros.

Piénsese en el ejemplo del proxy virtual de la sección de Motivación. La imagen debería cargarse en un determinado instante —en concreto, cuando se llama a la operación Dibujar—, y no cada vez que se hace referencia a la imagen. Sobrecargar el operador de acceso no permite realizar esta distinción. En ese caso debemos implementar manualmente cada operación del proxy que redirige la petición al sujeto.

Estas operaciones suelen ser muy parecidas unas a otras, como se demuestra en el Código de Ejemplo. Normalmente todas las operaciones verifican que la petición es legal, que existe el objeto original, etc., antes de redirigir la petición al sujeto. Escribir este código una y otra vez es una labor tediosa. Por ese motivo, es frecuente usar un preprocesador para generarlo automáticamente.

2. *Usar* doesNotUnderstand *en Smalltalk*. Smalltalk proporciona un enganche que se puede usar para permitir el reenvío automático de peticiones. Smalltalk llama a doesNotUnderstand: unMensaje cuando un cliente envía un mensaje a un receptor que no tiene el método correspondiente. La clase Proxy puede redefinir doesNotUnderstand para que el mensaje sea reenviado a su sujeto.

   Para garantizar que una petición se redirige al sujeto y no es absorbida en silencio por su proxy, puede definirse una clase Proxy que no entienda ningún mensaje. Smalltalk permite hacer esto definiendo Proxy como una clase sin superclase **[45]**.

   El principal inconveniente de doesNotUnderstand: es que la mayoría de sistemas Smalltalk tienen unos pocos mensajes especiales que son manejados directamente por la máquina virtual, y para éstos no se lleva a cabo la habitual búsqueda de métodos. El único que suele estar implementado en Object (y que puede por tanto afectar a los proxies) es la operación de identidad, ==.

   Si se decide usar doesNotUnderstand: para implementar Proxy será necesario hacer un diseño que tenga en cuenta este problema. No se debe suponer que la identidad entre proxies significa que sus sujetos reales son también idénticos. Un inconveniente añadido es que doesNotUnderstand: fue desarrollada para el tratamiento de errores, no para crear proxies, por lo que no suele ser demasiado rápida.

3. *Los proxies no siempre tienen que conocer el tipo del sujeto real*. Si una clase Proxy puede tratar con su sujeto sólo a través de una interfaz abstracta, entonces no hay necesidad de hacer una clase Proxy para cada clase de SujetoReal; el proxy puede tratar de manera uniforme a todas las clases SujetoReal. Pero si los objetos Proxy van a crear instancias de SujetoReal (como en el caso de los proxies virtuales), entonces tienen que conocer la clase

concreta.

Otra cuestión de implementación tiene que ver con cómo referirse al sujeto antes de que se cree una instancia de éste. Algunos proxies tienen que referirse a su sujeto independientemente de que esté en disco o en memoria. Eso significa que deben usar alguna forma de identificadores de objetos independientes del espacio de direcciones. En la sección de Motivación se usó un nombre de fichero para tal fin.

CÓDIGO DE EJEMPLO

El siguiente código implementa dos tipos de proxies: el proxy virtual descrito en la sección de Motivación y un proxy implementado con doesNotUnderstand **[46]**:

1. *Un proxy virtual.* La clase Grafico define la interfaz de los objetos gráficos:

```
class Grafico {
public:
     virtual ~Grafico();

     virtual    void    Dibujar(const
Puntos en) = 0;
     virtual                    void
ManejarRaton(Evento& evento) = 0;

     virtual       const       Punto&
ObtenerExtension() = 0;

     virtual   void   Cargar(istream&
desde) = 0;
     virtual   void   Guardar(ostream&
en) = 0;
protected:
     Grafico();
};
```

La clase Imagen implementa la interfaz Gráfico para mostrar ficheros de Imagen. Imagen redefine ManejarRaton para que los usuarios puedan cambiar

interactivamente el tamaño de la Imagen.

```
class Imagen : public Grafico {
public:
     Imagen(const char* fichero); //
carga una Imagen desde un fichero
     virtual ~Imagen));

     virtual    void    Dibujar(const
Punto& en);
     virtual                     void
ManejarRaton(Evento& evento);

     virtual       const       Punto&
ObtenerExtension();

     virtual   void   Cargar(istream&
desde);
     virtual   void   Guardar(ostream&
en);
private:
     // …
};
```

ProxyImagen tiene la misma interfaz que Imagen:

```
class ProxyImagen : public Grafico {
public:
     ProxyImagen(const          char*
FicheroDeImagen);
     virtual ~ProxyImagen();

     virtual    void    Dibujar(const
Punto& en);
     virtual                     void
ManejarRaton(Evento& evento);

     virtual       const       Puntos
ObtenerExtension();

     virtual   void   Cargar(istream&
desde);
     virtual   void   Guardar(ostream&
en);
protected:
```

```
        Imagen* ObtenerImagen();
private:
        Imagen* _imagen;
        Punto _extension;
        char* _nombreDeFichero;
};
```

El constructor guarda una copia local del nombre de fichero que contiene la Imagen, e inicializa_extension e _imagen:

```
ProxyImagen::ProxyImagen       (const
char* nombreDeFichero) {
        _nombreDeFichero               =
strdup(nombreDeFichero);
        _extension   =  Punto::Cero;   //
todavía no se conoce la extensión
        imagen =0;
}

Imagen* ProxyImagen::GetImagen() {
        if (_imagen == 0) {
                _imagen        =        new
Imagen(_nombreDeFichero);
        }
        return _imagen;
}
```

La implementación de ObtenerExtension devuelve, si es posible, la extensión guardada por el proxy; en otro caso se carga la Imagen desde el fichero. Dibujar carga la Imagen, y ManejarRaton reenvía el evento a la Imagen real.

```
const                          Punto&
ProxyImagen::ObtenerExtension () {
        if  (_extension  ==  Punto::Cero)
{
                _extension                =
ObtenerImagen()->ObtenerExtension();
        }
        return extension;
```

```
}

void   ProxyImagen::Dibujar   (const
Punto& en) {
     ObtenerImagen()->Dibujar(en);
}

void       ProxyImagen::ManejarRaton
(Evento& evento) {
     ObtenerImagen()-
>ManejarRaton(evento);
}
```

La operación Guardar graba en un flujo de salida la extensión y el nombre del fichero de Imagen almacenados en el proxy. Cargar recupera esta información e inicializa los miembros correspondientes.

```
void  ProxyImagen::Guardar  (ostream&
en) {
     en     <<     _extension     <<
_nombreDeFichero;
}

void  ProxyImagen::Cargar  (istream&
desde) {
     desde     »     _extension     »
_nombreDeFichero;
}
```

Por último, supongamos que tenemos una clase DocumentoDeTexto que puede contener objetos Grafico:

```
class DocumentoDeTexto {
public:
     DocumentoDeTexto();

     void Insertar(Grafico*);
     // …
};
```

Podemos insertar un ProxyImagen en un documento de texto como se muestra a continuación:

```
DocumentoDeTexto    texto    =    new
DocumentoDeTexto;
// …
texto->Insertar(new
ProxyImagen("nombreDeUnFicheroDeImagen"));
```

2. *Proxies que usan* doesNotUnderstand. Se pueden crear proxies genéricos en Smalltalk definiendo clases cuya superclase sea nil[47] y definiendo el método doesNotUnderstand: para manejar los mensajes.

   El siguiente método presupone que el proxy tiene un método sujetoReal que devuelve su sujeto real. En el caso de ProxyImagen, este método comprobaría si se había creado la Imagen, la crearía si fuese necesario y finalmente la devolvería. Hace uso de perform:withArguments: para responder al mensaje que ha sido atrapado en el sujeto real.

```
doesNotUnderstand: unMensaje
     ^ self sujetoReal
          perform:         unMensaje
selector
          withArguments:   unMensaje
arguments
```

   El argumento de doesNotUnderstand: es una instancia de Message que representa el mensaje que no entiende el proxy. Por tanto, el proxy responde a todos los mensajes asegurándose de que existe el sujeto real antes de reenviarle el mensaje.

   Una de las ventajas de doesNotUnderstand: es que puede realizar un procesamiento arbitrario. Por ejemplo, podríamos obtener un proxy de protección especificando

un conjunto mensajes Legales con los mensajes que deben ser aceptados y dándole al proxy el método siguiente:

```
doesNotUnderstand: unMensaje
    ^   (mensajesLegales   includes:
unMensaje selector)
          ifTrue: [self sujetoReal
               perform:    unMensaje
selector
               withArguments:
unMensaje arguments]
          ifFalse:    [self    error:
'Operador ilegal']
```

Este método comprueba que un mensaje sea legal antes de redirigirlo al sujeto real. En caso de que no lo sea, enviará error: al proxy, lo que daría como resultado un bucle infinito de errores a menos que el proxy defina error:. Por tanto, debería copiarse la definición de error: de la clase Object junto con cualquier método que use.

USOS CONOCIDOS

El proxy virtual de la sección de Motivación está tomado de las clases de ET++ para construcción de bloques de texto.

NEXTSTEP [Add94] usa proxies (instancias de la clase NXProxy) como proxies locales de objetos que pueden ser distribuidos. Un servidor crea proxies de objetos remotos cuando los solicitan los clientes. Al recibir un mensaje, el proxy lo codifica junto con sus argumentos y envía el mensaje codificado al sujeto remoto. De forma similar, el sujeto codifica los resultados a devolver y los envía de vuelta al objeto NXProxy.

McCullough [McC87] examinad uso de proxies en Smalltalk para accederá objetos remotos. Pascoe [Pas86] describe cómo proporcionar efectos laterales y control de acceso en las llamadas a métodos mediante “Encapsuladores”.

PATRONES RELACIONADOS

Adapter (131): un adaptador proporciona una interfaz diferente

para el objeto que adapta. Por el contrario, un proxy tiene la misma interfaz que su sujeto. No obstante, un proxy utilizado para protección de acceso podría rechazar una operación que el sujeto sí realiza, de modo que su interfaz puede ser realmente un subconjunto de la del sujeto.

Decorator (161): si bien los decoradores pueden tener una implementación parecida a los proxies, tienen un propósito diferente. Un decorador añade una o más responsabilidades a un objeto, mientras que un proxy controla él acceso a un objeto.

Los proxies difieren en el grado de similitud entre su implementación y la de un decorador. Un proxy de protección podría implementarse exactamente como un decorador. Por otro lado, un proxy remoto no contendrá una referencia directa a su sujeto real sino sólo una referencia indirecta, como "un ID de máquina y la dirección local en dicha máquina". Un proxy virtual empezará teniendo una referencia indirecta como un nombre de fichero, pero podrá al final obtener y utilizar una referencia directa.

DISCUSIÓN SOBRE LOS PATRONES ESTRUCTURALES

Tal vez haya notado similitudes entre los patrones estructurales, especialmente en sus participantes y colaboradores. Esto es debido a que los patrones estructurales se basan en un mismo pequeño conjunto de mecanismos del lenguaje para estructurar el código y los objetos: herencia simple y herencia múltiple para los patrones basados en clases, y composición de objetos para los patrones de objetos. Pero estas similitudes ocultan los diferentes propósitos de estos patrones. En esta sección se comparan y contrastan grupos de patrones estructurales para ofrecerle una visión de los méritos de cada uno de ellos.

ADAPTER FRENTE A BRIDGE

Los patrones Adapter (131) y Bridge (141) tienen algunos atributos comunes. Ambos promueven la flexibilidad al proporcionar un nivel de indirección a otro objeto. Ambos implican reenviar peticiones a este objeto desde una interfaz distinta de la suya propia.

La diferencia fundamental entre estos patrones radica en su propósito. El patrón Adapter se centra en resolver incompatibilidades entre dos interfaces existentes. No presta atención a cómo se implementan dichas interfaces, ni tiene en cuenta cómo podrían evolucionar de forma independiente. Es un modo de lograr que dos clases diseñadas independientemente trabajen juntas sin tener que volver a implementar una u otra. Por otro lado, el patrón Bridge une una implementación con sus implementaciones (que pueden ser numerosas). Proporciona una interfaz estable a los clientes permitiendo, no obstante, que cambien las clases que la implementan. También permite incorporar nuevas implementaciones a medida que evoluciona el sistema.

Como resultado de estas diferencias, los patrones Adapter y Bridge suelen usarse en diferentes puntos del ciclo de vida del

software. Un adaptador suele hacerse necesario cuando se descubre que deberían trabajar juntas dos clases incompatibles, generalmente para evitar duplicar código, y este acoplamiento no había sido previsto. Por el contrario, el usuario de un puente sabe de antemano que una abstracción debe tener varias implementaciones, y que una y otras pueden variar independientemente. El patrón Adapter hace que las cosas funcionen *después* de que han sido diseñadas; el Bridge lo hace *antes*. Eso no significa que el Adapter sea en modo alguno inferior al Bridge; simplemente, cada patrón resuelve un problema distinto.

Podemos ver una fachada (*véase* el patrón Facade (171)) como un adaptador para un conjunto de objetos. Pero esa interpretación obvia el hecho de que una fachada define una *nueva* interfaz, mientras que un adaptador reutiliza una interfaz existente. Recuérdese que un adaptador hace que trabajen juntas dos interfaces *existentes* en vez de tener que definir una completamente nueva.

COMPOSITE FRENTE A DECORATOR Y A PROXY

Los patrones Composite (151) y Decorator (161) tienen diagramas de estructura parecidos, lo que refleja el hecho de que ambos se basan en la composición recursiva para organizar un número indeterminado de objetos. Esta característica en común puede tentar a pensar en un objeto decorador como un compuesto degenerado, pero eso no representa la esencia del patrón Decorator. El parecido termina en la composición recursiva, al tratarse de nuevo de propósitos diferentes.

El patrón Decorator está diseñado para permitir añadir responsabilidades a objetos sin crear subclases. Esto evita la explosión de subclases a la que puede dar lugar al intentar cubrir cada combinación de responsabilidades estáticamente. El patrón Composite tiene un propósito diferente. Consiste en estructurar subclases para que se puedan tratar de manera uniforme muchos objetos relacionados, y que múltiples objetos puedan ser tratados como uno solo. Es decir, no se centra en la decoración sino en la representación.

Estos propósitos son distintos pero complementarios. Por tanto,

los patrones Composite y Decorador suelen usarse conjuntamente. Ambos llevan a la clase de diseño en la que se pueden construir aplicaciones simplemente poniendo juntos objetos sin definir ninguna clase nueva. Habrá una clase abstracta con algunas subclases que son compuestos, otras que son decoradores y otras que implementan los principales bloques de construcción del sistema. En este caso, tanto compuestos como decoradores tendrán una interfaz común. Desde el punto de vista del patrón Decorator, un compuesto es un ComponenteConcreto. Desde el punto de vista del patrón Composite, un decorador es una Hoja. Por supuesto, no tienen por qué ser usados juntos, y, como hemos visto, sus propósitos son bastante distintos.

Otro patrón con una estructura similar al Decorator es el Proxy (191). Ambos patrones describen cómo proporcionar un nivel de indirección a un objeto, y las implementaciones de los objetos proxy y decorador mantienen una referencia a otro objeto, al cual reenvían peticiones. Una vez más, no obstante, están pensados para propósitos diferentes.

Al igual que el Decorator, el patrón Proxy compone un objeto y proporciona una interfaz idéntica a los clientes. A diferencia del Decorator, el patrón Proxy no tiene que ver con asignar o quitar propiedades dinámicamente, y tampoco está diseñado para la composición recursiva. Su propósito es proporcionar un sustituto para un sujeto cuando no es conveniente o deseable acceder directamente a él, debido, por ejemplo, a residir en una máquina remota, tener acceso restringido o ser persistente.

En el patrón Proxy, el sujeto define la principal funcionalidad, y el proxy proporciona (o rechaza) el acceso al mismo. En el Decorator, el componente proporciona sólo parte de funcionalidad, y uno o más decoradores hacen el resto. El patrón Decorator se encarga de aquellas situaciones en las que no se puede determinar toda la funcionalidad de un objeto en tiempo de compilación, o al menos no resulta conveniente hacerlo. Ése no es el caso del Proxy, ya que éste se centra en una relación —entre el proxy y su sujeto— y dicha relación puede expresarse estáticamente.

Estas diferencias son significativas, ya que representan soluciones a problemas concretos y recurrentes en el diseño orientado a objetos. Pero eso no significa que estos patrones no

puedan combinarse. Podríamos pensar en un proxy-decorador que añadiese funcionalidad a un proxy, o en un decorador-proxy que adornase un objeto remoto. Si bien tales híbridos *pueden* ser útiles (no tenemos ejemplos reales a mano), pueden dividirse en patrones que *realmente* son útiles.

CAPÍTULO 5

Patrones de Comportamiento

CONTENIDO DEL CAPITULO

Los patrones de comportamiento tienen que ver con algoritmos y con la asignación de responsabilidades a objetos. Los patrones de comportamiento describen no sólo patrones de clases y objetos, sino también patrones de comunicación entre ellos. Estos patrones describen el flujo de control complejo que es difícil de seguir en tiempo de ejecución, lo que nos permite olvidarnos del flujo de control para concentrarnos simplemente en el modo en que se interconectan los objetos.

Los patrones de comportamiento basados en clases usan la herencia para distribuir el comportamiento entre clases. Este capítulo incluye dos de estos patrones. El patrón Template Method (229) es el más simple y común de los dos. Un método plantilla es una definición abstracta de un algoritmo. Define el algoritmo paso a paso, y cada paso invoca o bien a una operación abstracta o a una operación primitiva. Una subclase es la encargada de completar el algoritmo definiendo las operaciones abstractas. El otro patrón de comportamiento basado en clases es el Interpreter (225), que representa una gramática como una jerarquía de clases e implementa un intérprete como una operación sobre las instancias de dichas clases.

Los patrones de comportamiento basados en objetos usan la composición de objetos en vez de la herencia. Algunos describen cómo cooperan un grupo de objetos parejos para realizar una tarca que ningún objeto individual puede llevar a cabo por sí solo. Una cuestión importante es cómo los objetos saben unos de otros. Cada uno podría mantener referencias explícitas al resto, pero eso incrementaría su acoplamiento. Llevado al límite, cada objeto conocería a todos los demás. El patrón Mediator (251) evita esto introduciendo un objeto mediador entre todos los objetos parejos. El mediador proporciona la indirección necesaria para un bajo acoplamiento.

El patrón Chain of Responsibility (205) proporciona un

acoplamiento aún más bajo. Permite enviar peticiones a un objeto implícitamente, a través de una cadena de objetos candidatos. Cualquier candidato puede satisfacer la petición dependiendo de una serie de condiciones en tiempo de ejecución. El número de candidatos es indeterminado, y se puede seleccionar en tiempo de ejecución qué candidatos participan en la cadena.

El patrón Observer (269) define y mantiene una dependencia entre objetos. El ejemplo clásico de Observer tiene lugar en el Modelo/Vista/Controlador de Smalltalk, donde todas las vistas del modelo son avisadas cada vez que cambia el estado del mismo.

Otros patrones de comportamiento basados en objetos están relacionados con la encapsulación de comportamiento en un objeto, delegando las peticiones a dicho objeto. El patrón Strategy (289) encapsula un algoritmo en un objeto, facilitando especificar y cambiar el algoritmo que usa un objeto. El patrón Command (215) encapsula una petición en un objeto de modo que pueda ser pasada como parámetro, guardada en un historial o manipulada de alguna otra forma. El patrón State (279) encapsula los estados de un objeto para que éste pueda cambiar su comportamiento cuando cambia su estado. El Visitor (305) encapsula comportamiento que de otro modo estaría distribuido en varias clases, y el Iterator (237) abstrae el modo en que se accede y se recorren los objetos de una agregación.

CHAIN OF RESPONSABILITY

(Cadena de Responsabilidad)

PROPÓSITO

Evita acoplar el emisor de una petición a su receptor, dando a más de un objeto la posibilidad de responder a la petición. Encadena los objetos receptores y pasa la petición a través de la cadena hasta que es procesada por algún objeto.

MOTIVACIÓN

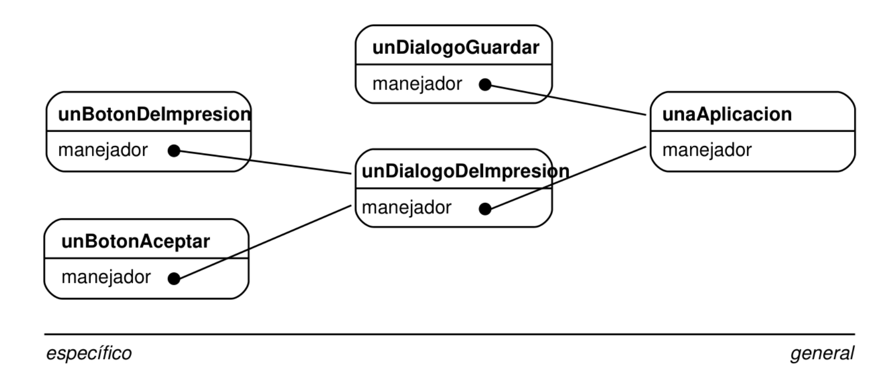
Supongamos un servicio de ayuda sensible al contexto para una interfaz gráfica de usuario. El usuario puede obtener información de ayuda en cualquier parte de la interfaz simplemente pulsando con el ratón sobre ella. La ayuda proporcionada depende de la parte de la interfaz que se haya seleccionado así como de su contexto; por ejemplo, un botón de un cuadro de diálogo puede tener diferente información de ayuda que un botón similar de la ventana principal. Si no existe información de ayuda específica para esa parte de la interfaz el sistema de ayuda debería mostrar un mensaje de ayuda más general sobre el contexto inmediato, por ejemplo, sobre d cuadro de diálogo en sí.

De ahí que sea natural organizar la información de ayuda de acuerdo con su generalidad —de lo más específico a lo más general —. Además, está claro que una petición de ayuda es manejada por un objeto entre varios de la interfaz de usuario; el objeto concreto depende del contexto y de la especificidad de la ayuda disponible.

El problema es que el objeto que en última instancia *proporciona* la ayuda no conoce explícitamente al objeto (por

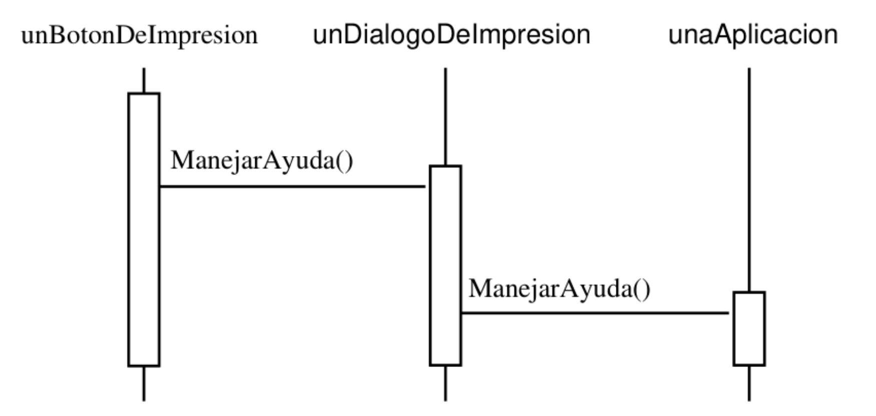
ejemplo, el botón) que *inicializa* la petición de ayuda. Necesitamos un modo de desacoplar el botón que da lugar a la petición de ayuda de los objetos que podrían proporcionar dicha información. El patrón Chain of Responsibility define cómo hacer esto.

La idea de este patrón es desacoplar a los emisores y a los receptores dándole a varios objetos la posibilidad de tratar una petición. La petición se pasa a través de una cadena de objetos hasta que es procesada por uno de ellos.



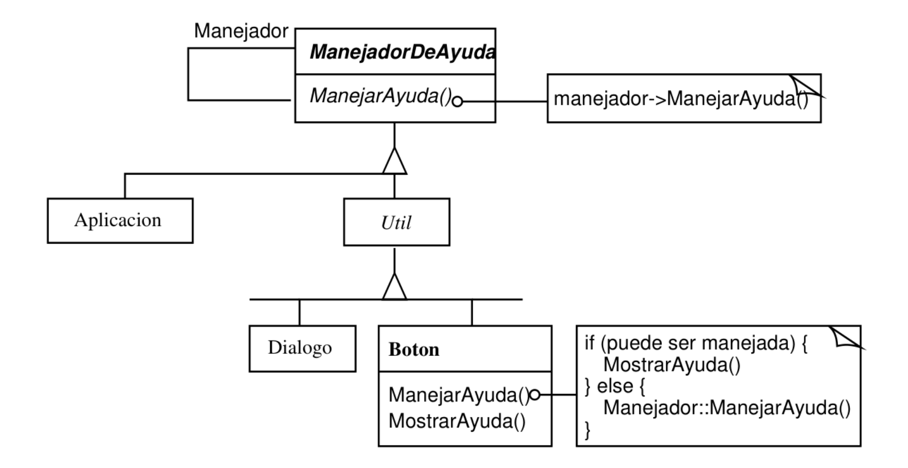
El primer objeto de la cadena recibe la petición y, o bien la procesa o bien la redirige al siguiente candidato en la cadena, el cual hace lo mismo. El objeto que hizo la petición no tiene un conocimiento explícito de quién la tratará —decimos que la petición tiene un **receptor implícito**—.

Supongamos que el usuario solicita ayuda sobre un botón denominado “Imprimir”. El botón se encuentra en una instancia de DialogoDeImpresion, que sabe a qué objeto de aplicación pertenece (*véase* el diagrama de objetos precedente). El siguiente diagrama de interacción ilustra cómo la petición de ayuda se reenvía a través de la cadena:



En este caso, la petición no es procesada ni por unBotonDeImpresion ni por unDialogoDeImpresion; se detiene en unaAplicacion, quien puede procesarla u obviarla. El cliente que dio origen a la petición no tiene ninguna referencia directa al objeto que finalmente la satisface.

Para reenviar la petición a lo largo de la cadena, y para garantizar que los receptores permanecen implícitos, cada objeto de la cadena comparte una interfaz común para procesar peticiones y para acceder a su **sucesor** en la cadena. Por ejemplo, el sistema de ayuda podría definir una clase ManejadorDeAyuda con su correspondiente operación Manejar Ayuda. ManejadorDeAyuda puede ser la clase padre de las clases de objetos candidatos, o bien puede ser definida como una clase mezclable. Entonces las clases que quieran manejar peticiones de ayuda pueden hacer que ManejadorDeAyuda sea uno de sus padres:



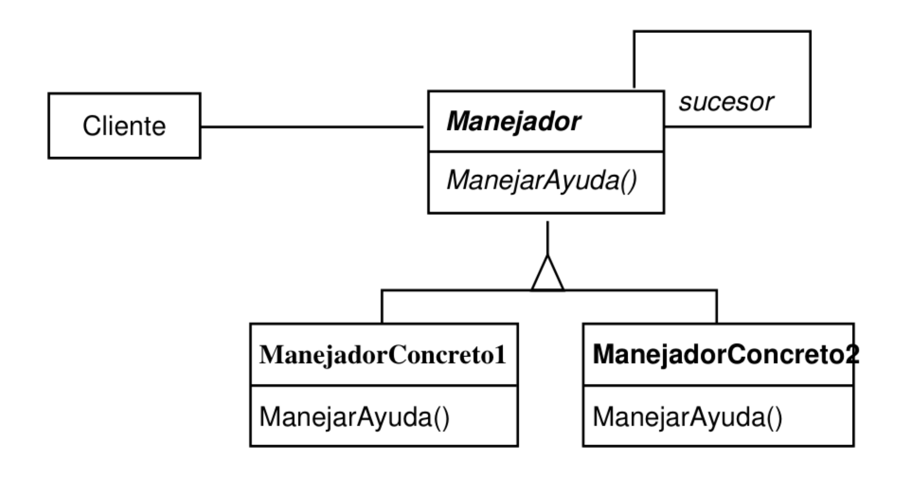
Las clases Boton, Dialogo y Aplicación usan las operaciones de ManejadorDeAyuda para tratar peticiones de ayuda. La operación ManejarAyuda de ManejadorDeAyuda reenvía la petición al sucesor de manera predeterminada. Las subclases pueden redefinir esta operación para proporcionar ayuda en determinadas circunstancias; en caso contrario, pueden usar la implementación predeterminada para reenviar la petición.

APLICABILIDAD

Úsese el patrón Chain of Responsibility cuando

- hay más de un objeto que pueden manejar una petición, y el manejador no se conoce *a priori,* sino que debería determinarse automáticamente
- se quiere enviar una petición a un objeto entre varios sin especificar explícitamente el receptor.
- el conjunto de objetos que pueden tratar una petición debería ser especificado dinámicamente.

ESTRUCTURA



Una estructura de objetos típica podría parecerse a ésta:



PARTICIPANTES

- **Manejador** (ManejadorDeAyuda)
  - define una interfaz para tratar las peticiones.
  - (opcional) implementa el enlace al sucesor.
- **ManejadorConcreto** (BotonDeImpresion, DialogoDeImpresion)
  - trata las peticiones de las que es responsable.
  - puede acceder a su sucesor.
  - si el ManejadorConcreto puede manejar la petición, lo hace; en caso contrario la reenvía a su sucesor.
- **Cliente**
  - inicializa la petición a un objeto ManejadorConcreto de la cadena.

COLABORACIONES

Cuando un cliente envía una petición, ésta se propaga a través de la

cadena hasta que un objeto ManejadorConcreto se hace responsable de procesarla.

CONSECUENCIAS

Este patrón tiene las siguientes ventajas e inconvenientes:

1. *Reduce el acoplamiento*. El patrón libera a un objeto de tener que saber qué otro objeto maneja una petición. Un objeto sólo tiene que saber que una petición será manejada "de forma apropiada". Ni el receptor ni el emisor se conocen explícitamente entre ellos, y un objeto de la cadena tampoco tiene que conocer la estructura de ésta. Como resultado, la Cadena de Responsabilidad puede simplificar las interconexiones entre objetos. En vez de que los objetos mantengan referencias a todos los posibles receptores, sólo tienen una única referencia a su sucesor.
2. *Añade flexibilidad para asignar responsabilidades a objetos*. La Cadena de Responsabilidad ofrece una flexibilidad añadida para repartir responsabilidades entre objetos. Se pueden añadir o cambiar responsabilidades para tratar una petición modificando la cadera en tiempo de ejecución. Esto se puede combinar con la herencia para especializar los manejadores estáticamente.
3. *No se garantiza la recepción*. Dado que las peticiones no tienen un receptor explícito, no hay *garantía* de que sean manejadas —la petición puede alcanzar el final de la cadena sin haber sido procesada—. Una petición también puede quedar sin tratar cuando la cadena no está configurada correctamente.

IMPLEMENTACIÓN

Éstos son algunos detalles de implementación a tener en cuenta sobre la Cadena de Responsabilidad:

1. *Implementación de la cadena sucesora*. Hay dos formas posibles de implementar la cadena sucesora:

- Definir nuevos enlaces (normalmente en el Manejador, pero también podría ser en los objetos ManejadorConcreto).
- Usar los enlaces existentes.

Los ejemplos mostrados hasta ahora definen nuevos enlaces, pero muchas veces se pueden usar referencias a objetos existentes para formar la cadena sucesora. Por ejemplo, las referencias al padre en una jerarquía de parte-todo pueden definir el sucesor de una parte. Una estructura de útiles **[48]** puede que ya tenga dichos enlaces El patrón Composite (151) describe las referencias al padre con más detalle.

Usar enlaces existentes funciona bien cuando los enlaces permiten la cadena que necesitamos. Nos evita tener que definir explícitamente nuevos enlaces y ahorra espacio. Pero si la estructura no refleja la cadena de responsabilidad que necesita nuestra aplicación habrá que definir enlaces redundantes.

2. *Conexión de los sucesores*. Si no hay referencias preexistentes para definir una cadena, entonces tendremos que introducirlas nosotros mismos. En ese caso, el Manejador no sólo define la interfaz para las peticiones, sino que normalmente también se encarga de mantener el sucesor. Eso permite que el manejador proporcione una implementación predeterminada de ManejarPeticion que reenvíe la petición al sucesor (si hay alguno). Si una subclase de ManejadorConcreto no está interesada en dicha petición, no tiene que redefinir la operación de reenvío, puesto que la implementación predeterminada la reenvía incondicionalmente.

   A continuación se muestra una clase base ManejadorDeAyuda que mantiene un enlace al sucesor:

```
class ManejadorDeAyuda {
public:
     ManejadorDeAyuda(ManejadorDeAyuda*
s) : _sucesor(s) { }
```

```
     virtual void ManejarAyuda();
private:
     ManejadorDeAyuda* _sucesor;
};

void  ManejadorDeAyuda::ManejarAyuda
() {
     if (_sucesor) {
          _sucesor->ManejarAyuda();
     }
}
```

3. *Representación de las peticiones*. Hay varias opciones para representar las peticiones. En su forma más simple, una petición es una invocación a una operación insertada en el código, como en el caso de ManejarAyuda. Esto resulta conveniente y seguro, pero entonces sólo se pueden reenviar el conjunto prefijado de peticiones que define la clase Manejador.

   Una alternativa es usar una única función manejadora que reciba un código de petición (por ejemplo, una constante entera o una cadena) como parámetro. Esto permite un número arbitrario de peticiones. El único requisito es que el emisor y el receptor se pongan de acuerdo sobre cómo debe codificarse la petición.

   Este enfoque es más flexible, pero requiere sentencias condicionales para despachar la petición en función de su código. Y, lo que es peor, no hay un modo de pasar los parámetros seguro con respecto al tipo, por lo que éstos deben ser empaquetados y desempaquetados manualmente. Obviamente, esto es menos seguro que invocar una operación directamente.

   Para resolver el problema del paso de parámetros, podemos usar para las peticiones *objetos* aparte que incluyan los parámetros de la petición. Una clase Petición puede representar peticiones explícitamente, y se pueden definir nuevos tipos de peticiones mediante herencia. Las subclases pueden definir diferentes parámetros. Los

manejadoras deben conocer el tipo de petición (esto es, qué subclase de Petición están usando) para acceder a estos parámetros.

Para identificar la petición. Petición puede definir una función de acceso que devuelva un identificador para la clase. Por otro lado, el receptor puede usar información de tipos en tiempo de ejecución en caso de que el lenguaje de implementación lo permita.

A continuación se muestra un esbozo de una función de despacho que usa objetos petición para identificar las peticiones. Una operación ObtenerTipo definida en la clase base Petición identifica el tipo de petición:

```
void       Manejador::ManejarPeticion
(Peticion* laPeticion) {
     switch               (laPeticion-
>ObtenerTipo()) {
     case Ayuda:
          // convierte el argumento
al tipo apropiado
          ManejarAyuda((PeticionDeAyuda*)
laPeticion);
          break;

     case Impresion:
          ManejarImpresion!
(PeticionDeImpresion*) laPeticion);
          // …
          break;

     default:
          // …
          break;
     }
}
```

Las subclases pueden extender el despacho «definiendo Manejarpeticion. La subclase maneja sólo aquellas peticiones en las que está interesada; otras peticiones son reenviadas a la clase padre. De esta forma, las subclases

efectivamente extienden (en vez de redefinir) la operación Manejarpeticion. Por ejemplo, así es como una subclase ManejadorExtendido extiende la versión de Manejarpeticion de Manejador:

```
class ManejadorExtendido : public
Manejador {
public:
     virtual void
ManejarPeticion(Peticion*
laPeticion);
     // …
};

void
ManejadorExtendido::Manejarpeticion
(Peticion* laPeticion) {
     switch (laPeticion-
>ObtenerTipo()) {
     case VistaPreliminar:
          // trata la operación
VistaPreliminar
          break;

     default:
          // permite que Manejador
trate otras peticiones
          Manejador::ManejarPeticion(laPeticion);
     }
}
```

4. *Reenvío automático en Smalltalk*. Podemos usar el mecanismo de Smalltalk doesNotUnderstand para reenviar peticiones. Los mensajes que no tienen su método correspondiente son atrapados en la implementación de doesNotUnderstand, que puede ser redefinida para reenviar el mensaje a un sucesor del objeto. De esa manera no es necesario implementar manualmente el reenvío; la clase maneja sólo la petición en la que está interesada, y deja a doesNotUnderstand el reenvío de todas las demás.

CÓDIGO DE EJEMPLO

El siguiente ejemplo ilustra cómo una cadena de responsabilidad puede manejar peticiones para un sistema de ayuda en línea como el descrito anteriormente. La petición de ayuda es una operación explícita. Usaremos las referencias al padre existentes en la jerarquía de útiles para propagar peticiones entre útiles de la cadena, y definiremos una referencia en la clase Manejador para propagarlas peticiones de ayuda entre los elementos de la cadena que no sean útiles.

La clase ManejadorDeAyuda define la interfaz para manejar peticiones de ayuda. Mantiene un tema de ayuda (que, de manera predeterminada, siempre está vacío) y guarda una referencia a su sucesor en la cadena de manejadores de ayuda. La operación principal es ManejarAyuda, la cual es redefinida por las subclases. TieneAyuda es una operación de conveniencia para comprobar si existe un tema de ayuda asociado.

```
typedef int Tema;
     const Tema SIN_TEMA_DE_AYUDA = -1;

     class ManejadorDeAyuda {
     public:
          ManejadorDeAyuda(ManejadorDeAyuda*
= 0, Tema = SIN_TEMA_DE_AYUDA);
          virtual bool TieneAyuda();
          virtual                        void
EstablecerManejador(ManejadorDeAyuda*, Tema);
          virtual void ManejarAyuda();
     private:
          ManejadorDeAyuda* _sucesor;
          Tema _tema;
     };

     ManejadorDeAyuda::ManejadorDeAyuda (
          ManejadorDeAyuda* m, Tema t
     ) : _sucesor(m), _tema(t) { }

     bool ManejadorDeAyuda::TieneAyuda () {
          return _tema != SIN_TEMA_DE_AYUDA;
     }

     void ManejadorDeAyuda::ManejarAyuda () {
     if (_sucesor != 0) {
          _sucesor->ManejarAyuda();
     }
```

```
}
```

Todos los útiles son subclases de la clase abstracta Util. Util es una subclase de ManejadorDeAyuda, ya que todos los elementos de la interfaz de usuario pueden tener ayuda asociada a ellos (también podríamos haber usado una implementación basada en una clase mezclable).

```
class Util : public ManejadorDeAyuda {
protected:
          Util(Util*   padre,   Tema   t   =
SIN_TEMA_DE_AYUDA);
private:
     Util* _padre;
};

Util::Util    (Util*    u,    Tema    t)    :
ManejadorDeAyuda(u, t) {
     _padre = u;
}
```

En nuestro ejemplo, un botón es el primer manejador de la cadena. La clase Boton es una subclase de Util. El constructor de Boton toma dos parámetros: una referencia al útil que lo contiene y el tema de ayuda.

```
class Boton : public Util {
public:
     Boton(Util*     d,     Tema     t     =
SIN_TEMA_DE_AYUDA);

     virtual void ManejarAyuda();
     // Las operaciones de Util redefinidas
por Boton…
};
```

La versión de ManejarAyuda de Boton en primer lugar comprueba si hay un tema de ayuda para los botones. Si el desarrollador no ha definido ninguno, entonces la petición es reenviada al sucesor

usando la operación ManejarAyuda de ManejadorDeAyuda. Si *hay* un tema de ayuda el botón la muestra y termina la búsqueda.

```
Boton::Boton (Util* h, Tema t) : Util(h, t) {
}

void Boton::ManejarAyuda () {
     if (TieneAyuda()) {
          // ofrecer ayuda sobre el botón
     } else {
          ManejadorDeAyuda::ManejarAyuda();
     }
}
```

Dialogo implementa un esquema similar, salvo que su sucesor no es un útil, sino *cualquier* manejador de ayuda. En nuestra aplicación este sucesor será una instancia de Aplicación.

```
class Dialogo : public Util {
public:
     Dialogo(ManejadorDeAyuda* m, Tema t =
SIN_TEMA_DE_AYUDA);
     virtual void ManejarAyuda!);

     // las operaciones de Util redefinidas
por Dialogo…
     // …
};

Dialogo::Dialogo (ManejadorDeAyuda* m, Tema
t) : Util(0) {
     EstablecerManejador(m, t);
}

void Dialogo::ManejarAyuda () {
     if (TieneAyuda()) {
          // ofrecer ayuda sobre el diálogo
     } else (
          ManejadorDeAyuda::ManejarAyuda();
     }
}
```

Al final de la cadena hay una instancia de Aplicación. La aplicación no es un útil, por lo que Aplicación hereda directamente de ManejadorDeAyuda. Cuando una petición de ayuda se propaga hasta este nivel, la aplicación puede proporcionar información sobre la aplicación en general, o puede ofrecer una lista con los distintos temas de ayuda:

```
class Aplicacion : public ManejadorDeAyuda {
public:
     Aplicación(Tema t) : ManejadorDeAyuda(0,
t) { }

virtual void ManejarAyuda();
     //   operaciones   específicas   de   la
aplicación…
};

void Aplicación::ManejarAyuda () {
     // muestra una lista de temas de ayuda
}
```

El siguiente código crea estos objetos y los conecta. En este caso el diálogo es sobre la impresión, y por tanto los objetos tienen asignados temas relacionados con la impresión.

```
const Tema TEMA_IMPRESION = 1;
const Tema TEMA_ORIENTACION_PAPEL = 2;
const Tema TEMA_APLICACION = 3;

Aplicacion*      aplicacion      =      new
Aplicacion(TEMA_APLICACION);
Dialogo*  dialogo  =  new  Dialogo(aplicacion,
TEMA_IMPRESION);
Boton*    boton    =    new    Boton(dialogo,
TEMA_ORIENTACION_PAPEL);
```

Podemos invocar a la petición de ayuda llamando a ManejarAyuda en cualquier objeto de la cadena. Para comenzar la búsqueda en el objeto botón basta con llamar a ManejarAyuda sobre él:

```
        boton->ManejarAyuda();
```

En este caso, el botón manejará la petición inmediatamente. Nótese que cualquier clase ManejadorDeAyuda podría ser el sucesor de Dialogo. Más aún, podría cambiarse dinámicamente su sucesor. De modo que no importa dónde se use un diálogo, siempre se obtendrá la información de ayuda dependiente del contexto apropiada para él.

USOS CONOCIDOS

Varias bibliotecas de clases usan el patrón Chain of Responsibility para manejar los eventos de usuario. Aunque usan distintos nombres para la clase Manejador, la idea es la misma: cuando el usuario hace clic con el ratón o pulsa una tecla, se genera un evento y se pasa a lo largo de la cadena. MacApp [App89] y ET++ [WGM88] lo llaman "EventHandler" (manejador de eventos) la biblioteca TCL de Symantec [Sym93b] lo llama "Bureaucrat" (burócrata) y AppKit de NeXT [Add94] usa el nombre "Responder" (respondedor).

El framework de editores gráficos Unidraw define objetos Command que encapsulan peticiones a los objetos Component y ComponentView [VL90], Las órdenes son peticiones en el sentido de que un componente o una vista de un componente pueden interpretar una orden para realizar una operación. Esto se corresponde con el enfoque de "peticiones como objetos" descrito en la sección de Implementación. Los componentes y las vistas de componentes se pueden estructurar jerárquicamente. Un componente o una vista de componente pueden reenviar interpretaciones de órdenes a su padre, quien a su vez puede reenviarlas a su padre y así sucesivamente, formando así una cadena de responsabilidad.

ET++ usa una Cadena de Responsabilidad para tratar la actualización de gráficos. Un objeto gráfico llama a la operación InvalidateRect cada vez que debe actualizarse una parte de su representación. Un objeto gráfico no puede manejar InvalidateRect él mismo, ya que no sabe lo suficiente sobre su contexto. Por ejemplo, un objeto gráfico puede formar parte de objetos como

barras de desplazamiento o *zooms* que transforman su sistema de coordenadas. Eso significa que podemos desplazamos o hacer *zoom* sobre el objeto, de manera que éste quede parcialmente oculto. Por tanto la implementación predeterminada de InvalidateRect reenvía la petición al objeto contenedor. El último objeto de la cadena de reenvío es una instancia de Window. En el momento en que Window recibe la petición, se garantiza que el rectángulo de invalidación se transforma correctamente. El objeto Window trata InvalidateRect notificando a la interfaz del sistema de ventanas y solicitando actualizarse.

PATRONES RELACIONADOS

Este patrón se suele aplicar conjuntamente con el patrón Composite (151). En él, los padres de los componentes pueden actuar como sucesores.

COMMAND (Orden)

PROPÓSITO

Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con diferentes peticiones, hacer cola o llevar un registro de las peticiones, y poder deshacer las operaciones.

TAMBIÉN CONOCIDO COMO

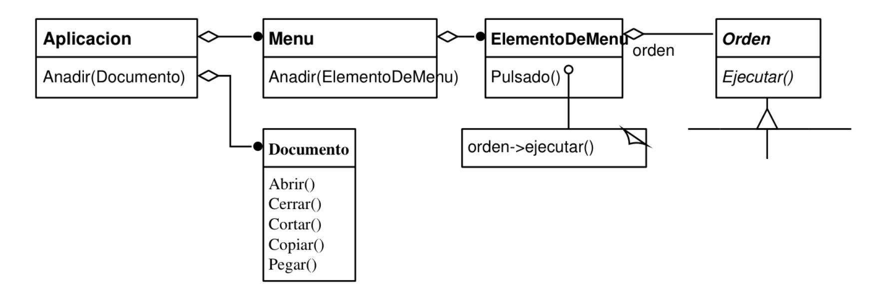
Action (Acción), *Transaction* (Transacción)

MOTIVACIÓN

A veces es necesario enviar peticiones a objetos sin saber nada acerca de la operación solicitada o de quién es el receptor de la petición. Por ejemplo, los toolkits de interfaces de usuario incluyen objetos como botones y menús que realizan una petición en respuesta a una entrada de usuario. Pero el toolkit no puede implementar la petición explícitamente en el botón o el menú, ya que sólo las aplicaciones que usan el toolkit saben qué debería hacerse y sobre qué objeto. Como diseñadores de toolkits no tenemos modo de conocer al receptor de la petición ni de saber qué operaciones se efectuarán.

El patrón Command permite que los objetos del toolkit hagan peticiones a objetos de la aplicación no especificados, convirtiendo a la propia petición en un objeto, el cual se puede guardar y enviar exactamente igual que cualquier otro objeto. La clave de este patrón es una clase abstracta Orden, que declara una interfaz para ejecutar operaciones. En su forma más simple, esta interfaz incluye una operación abstracta Ejecutar. Las subclases concretas de Orden especifican un par receptor-acción, guardando el receptor como una variable de instancia e implementando Ejecutar para que invoque a la petición. El receptor posee el conocimiento necesario para llevar

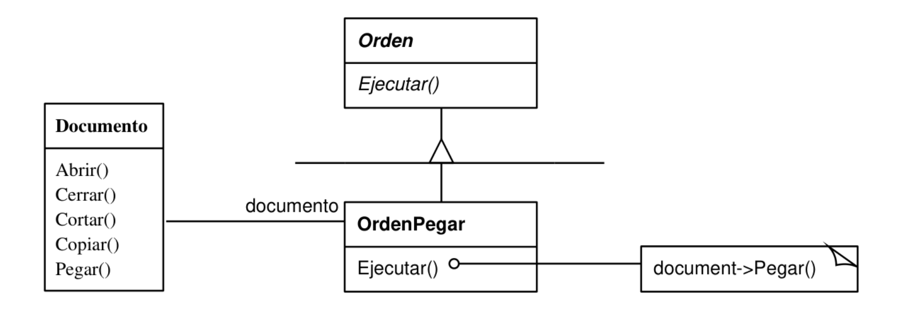
a cabo la petición.



Los menús se pueden implementar fácilmente con objetos Orden. Cada opción de un Menu es una instancia de una clase ElementoDeMenu. La clase Aplicación es la encargada de crear estos menús y sus elementos de menú junto con el resto de la interfaz de usuario. También es esta clase quien sabe qué objetos Documento han sido abiertos por un usuario.

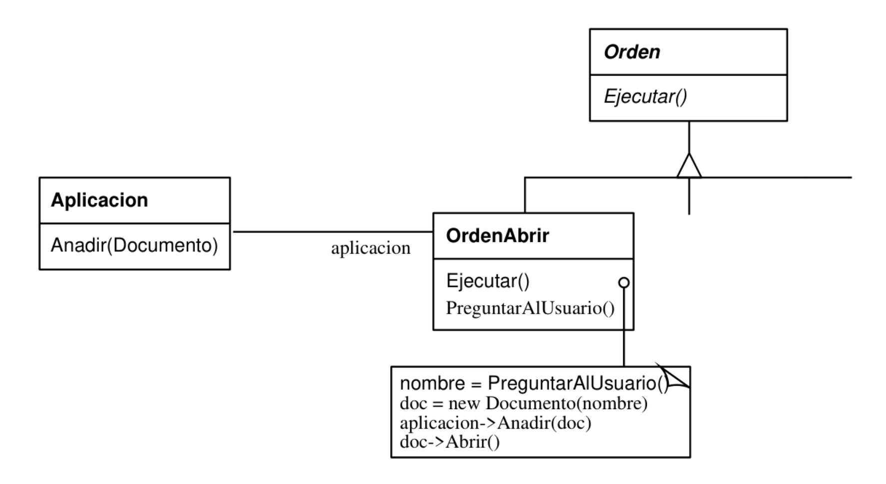
La aplicación configura cada ElementoDeMenu con una instancia de una subclase concreta de Orden. Cuando el usuario selecciona un ElementoDeMenu, éste llama al método Ejecutar de su orden quien lleva a cabo la operación. Los objetos ElementoDeMenu no saben qué subclases de Orden usan! Las subclases de Orden almacenan el receptor de la petición e invocan sobre él una o más operaciones.

Por ejemplo, OrdenPegar permite pegar texto del portapapeles en un Documento. El receptor de OrdenPegar es el objeto Documento proporcionado cuando se crea una instancia de aquélla. La operación Ejecutar llama a Pegar sobre el Documento receptor.

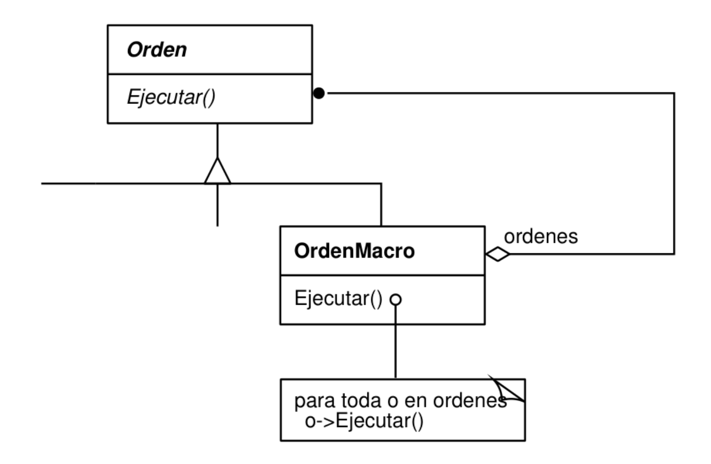


La operación Ejecutar de OrdenAbrir es diferente: le pregunta al usuario el nombre de un documento, crea el correspondiente objeto

Documento, lo añade a la aplicación receptora y abre el documento.



A veces un ElementoDeMenu necesita ejecutar una *secuencia* de órdenes. Por ejemplo, podría construirse un ElementoDeMenu para centrar una página a tamaño normal a partir de un objeto OrdenCentrarDocumento y de otro objeto OrdenTamanoNormal. Dado que es habitual concatenar órdenes de este modo, podemos definir una clase OrdenMacro para permitir que un ElementoDeMenu ejecute un número indefinido de órdenes. OrdenMacro es una subclase concreta de Orden que simplemente ejecuta una secuencia de órdenes. OrdenMacro no tiene ningún receptor explícito, sino que son las órdenes que contiene las que definen su propio receptor.



Nótese cómo, en cada uno de los ejemplos anteriores, el patrón Orden desacopla el objeto que invoca la operación de aquél que posee el conocimiento para realizarla. Esto nos da mucha flexibilidad para diseñar nuestra interfaz de usuario. Una aplicación puede proporcionar un menú y un botón como interfaces para una misma función simplemente haciendo que ambos compartan una instancia de la misma subclase concreta de Orden. Podemos reemplazar órdenes dinámicamente, lo que será útil para implementar menús sensibles al contexto. También podemos permitir la creación de órdenes mediante la composición de unas órdenes en otras más grandes. Todo esto es posible debido a que el objeto que emite la petición sólo necesita saber cómo enviarla; no necesita saber cómo se ejecutará la petición.

APLICABILIDAD

Úsese el patrón Command cuando se quiera

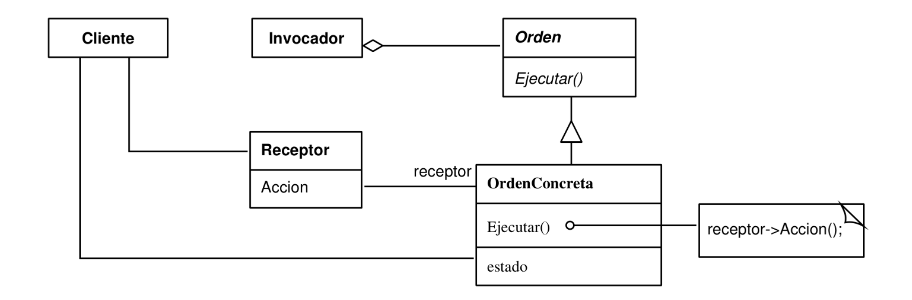
- parametrizar objetos con una acción a realizar, como ocurría con los objetos ElementoDeMenu anteriores. En un lenguaje de procedimiento se puede expresar dicha parametrización con una función ***callback***, es decir, con una función que está registrada en algún sitio para que sea llamada más tarde. Los objetos Orden son un sustituto orientado a objetos para las funciones callback.
- especificar, poner en cola y ejecutar peticiones en diferentes instantes de tiempo. Un objeto Orden puede tener un tiempo de vida independiente de la petición original. Si se puede representar el receptor de una petición en una forma independiente del espacio de direcciones, entonces se puede transferir un objeto orden con la petición a un proceso diferente y llevar a cabo la petición allí.
- permitir deshacer. La operación Ejecutar de Orden puede guardar en la propia orden el estado que anule sus efectos. Debe añadirse a la interfaz Orden una operación Deshacer que anule los efectos de una llamada anterior a Ejecutar. Las órdenes ejecutadas se guardan en una lista que hace las veces de historial. Se pueden lograr niveles ilimitados de deshacer y repetir recorriendo dicha lista hacia atrás y hacia delante

llamando respectivamente a Deshacer y Ejecutar.

- permitir registrar los cambios de manera que se puedan volver a aplicar en caso de una caída del sistema. Aumentando la interfaz de Orden con operaciones para cargar y guardar se puede mantener un registro persistente de los cambios. Recuperarse de una caída implica volver a cargar desde el disco las órdenes guardadas y volver a ejecutarlas con la operación Ejecutar.
- estructurar un sistema alrededor de operaciones de alto nivel construidas sobre operaciones básicas. Dicha estructura es común en los sistemas de información que permiten **transacciones**.

Una transacción encapsula un conjunto de cambios sobre unos datos. El patrón Command ofrece un modo de modelar transacciones. Las órdenes tienen una interfaz común, permitiendo así invocar a todas las transacciones del mismo modo. El patrón también facilita extender el sistema con nuevas transacciones.

ESTRUCTURA



PARTICIPANTES

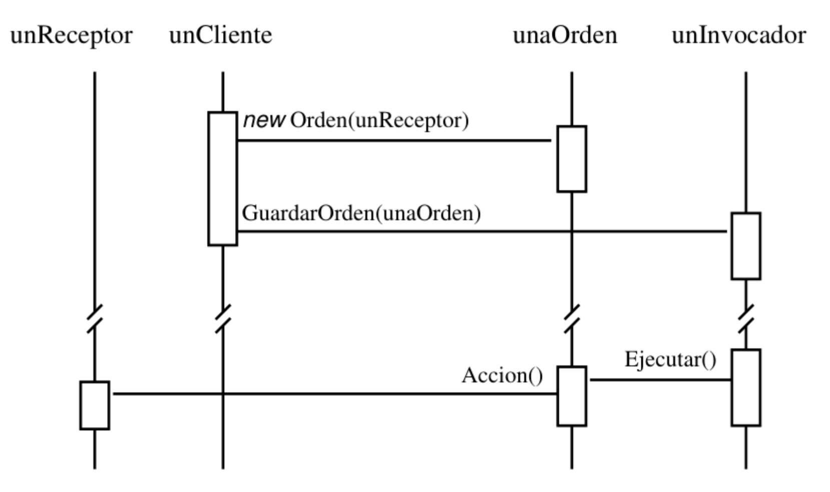
- **Orden**
  - declara una interfaz para ejecutar una operación.
- **OrdenConcreta** (OrdenPegar, OrdenAbrir)
  - define un enlace entre un objeto Receptor y una acción.
  - implementa Ejecutar invocando la correspondiente operación u operaciones del Receptor.

- **Cliente** (Aplicación)
    - crea un objeto OrdenConcreta y establece su receptor.
- **Invocador** (ElementoDeMenu)
    - le pide a la orden que ejecute la petición.
- **Receptor** (Documento, Aplicación)
    - sabe cómo llevar a cabo las operaciones asociadas a una petición. Cualquier clase puede hacer actuar como Receptor.

COLABORACIONES

- El cliente crea un objeto OrdenConcreta y especifica su receptor.
- Un objeto Invocador almacena el objeto OrdenConcreta.
- El invocador envía una petición llamando a Ejecutar sobre la orden. Cuando las órdenes se pueden deshacer, OrdenConcreta guarda el estado para deshacer la orden antes de llamar a Ejecutar.
- El objeto OrdenConcreta invoca operaciones de su receptor para llevar a cabo la petición.

El siguiente diagrama muestra las interacciones entre estos objetos, ilustrando cómo Orden desacopla el invocador del receptor (y de la petición que éste lleva a cabo).



CONSECUENCIAS

El patrón Command tiene las siguientes consecuencias:

1. Orden desacopla el objeto que invoca la operación de aquél que sabe cómo realizarla.
2. Las órdenes son objetos de primera clase. Pueden ser manipulados y extendidos como cualquier otro objeto.
3. Se pueden ensamblar órdenes en una orden compuesta. Un ejemplo lo constituye la clase OrdenMacro que se describió antes. En general, las órdenes compuestas son una instancia del patrón Composite (151).
4. Es fácil añadir nuevas órdenes, ya que no hay cambiar las clases existentes.

IMPLEMENTACIÓN

A la hora de implementar el patrón Command deben tenerse en cuenta las siguientes cuestiones:

1. ¿*Cómo debería ser de inteligente una orden*? Una orden puede tener un amplio conjunto de habilidades. Por un lado, simplemente define un enlace entre un receptor y las acciones que lleva a cabo la petición. Por el otro, lo implementa todo ella misma sin delegar para nada en el

receptor. Este último extremo resulta útil cuando queremos definir órdenes que sean independientes de las clases existentes, cuando no existe ningún receptor adecuado o cuando una orden conoce implícitamente a su receptor. Por ejemplo, una orden que crea otra ventana de aplicación puede ser tan capaz de crear la ventana como cualquier otro objeto. En algún punto entre estos dos extremos se encuentran las órdenes que tienen el conocimiento suficiente para encontrar dinámicamente sus receptores.

2. *Permitir deshacer y repetir*. Las órdenes pueden permitir capacidades de deshacer y repetir sí proveen un modo de revertir su ejecución (por ejemplo, mediante una operación Deshacer). Una clase OrdenConcreta podría necesitar almacenar información de estado adicional para hacer esto. Este estado puede incluir

- el objeto Receptor, el cual es quien realmente realiza las operaciones en respuesta a la petición;
- los argumentos de la operación llevada a cabo por el receptor; y
- cualquier valor original del receptor que pueda cambiar como resultado de manejar la petición. El receptor debe proporcionar operaciones que permitan a la orden devolver el receptor a su estado anterior.

Para permitir un nivel de deshacer, una aplicación sólo necesita guardar la última orden que se ejecutó. Para múltiples niveles de deshacer y repetir, la aplicación necesita un historial de las órdenes que han sido ejecutadas, donde la máxima longitud de la lista determina el número de niveles de deshacer/repetir. El historial guarda secuencias de órdenes que han sido ejecutadas. Recorrer la lista hacia atrás deshaciendo las órdenes cancela sus efectos; recorrerla hacia delante ejecutando las órdenes los repite.

Una orden anulable puede que deba ser copiada antes de

que se guarde en el historial. Eso es debido a que el objeto orden que llevó a cabo la petición original desde, supongamos, un ElementoDeMenu, más tarde ejecutará otras peticiones. La copia es necesaria para distinguir entre diferentes invocaciones de la misma orden si su estado puede variar entre invocaciones sucesivas. Por ejemplo, una OrdenBorrar que borra los objetos seleccionados debe guardar diferentes conjuntos de objetos cada vez que se ejecuta. Por tanto el objeto OrdenBorrar deberá ser copiado después de su ejecución y esta copia almacenada en el historial. En caso de que el estado de la orden no cambie tras su ejecución no es necesario realizar la copia, basta con guardar en el historial una referencia a la orden. Las órdenes que deben ser copiadas antes de ser guardadas en el historial funcionan como prototipos (véase el patrón Prototype (109)).

Evitar la acumulación de errores en el proceso de deshacer. La histéresis puede ser un problema a la hora de garantizar un mecanismo de deshacer/repetir fiable, que preserve la semántica. Los errores se pueden acumular a medida que se ejecutan y deshacen las órdenes repetidamente, de manera que el estado de una aplicación finalmente difiera de sus valores originales. Por tanto, puede ser necesario guardar más información con la orden para asegurar que los objetos son devueltos a su estado original. Puede aplicarse el patrón Memento (261) para dar a la orden acceso a esta información sin exponer las interioridades de otros objetos.

Uso de plantillas de C++. Para aquellas órdenes que (1) no se pueden deshacer y (2) no necesitan argumentos, podemos usar plantillas de C++ para evitar crear una subclase de Orden para cada clase de acción y receptor. Mostraremos cómo hacer esto en la sección de Código de Ejemplo.

CÓDIGO DE EJEMPLO

El código de C++ que se muestra aquí es un esbozo de las clases

Orden que se comentaron en la sección de Motivación. Definiremos las clases OrdenAbrir, OrdenPegar y OrdenMacro. Veamos en primer lugar la clase abstracta Orden:

```
class Orden {
public:
     virtual ~Orden();

     virtual void Ejecutar() = 0;
protected:
     Orden();
};
```

OrdenAbrir abre un documento cuyo nombre es proporcionado por el usuario. Es necesario pasarle un objeto Aplicación en su constructor. PreguntarAlUsuario es una rutina de implementación que le pide al usuario el nombre del documento a abrir.

```
class OrdenAbrir : public Orden {
public:
     OrdenAbrir(Aplicacion*);

     virtual void Ejecutar();
protected:
     virtual            const            char*
PreguntarAlUsuario();
private:
     Aplicacion* _aplicacion;
     char* _respuesta;
};

OrdenAbrir::OrdenAbrir (Aplicación* a) {
     _aplicacion = a;
}

void OrdenAbrir::Ejecutar () {
     const       char*       nombre        =
PreguntarAlUsuario();

     if (nombre 1= 8) {
          Documento*    documento    =    new
Documento(nombre);
          _aplicacion->Anadir(documento);
          documento->Abrir();
```

```
          }
     }
```

A OrdenPegar es necesario pasarle un objeto Documento como su receptor. El receptor se pasa como parámetro en el constructor de OrdenPegar.

```
class OrdenPegar : public Orden {
public:
     OrdenPegar(Documento*);

     virtual void Ejecutar();
private:
     Documento* _documento;
};

OrdenPegar::OrdenPegar (Documento* doc) {
     _documento = doc;
}

void OrdenPegar::Ejecutar () {
     documento->Pegar();
}
```

Para órdenes simples que no se pueden deshacer y que no necesitan argumentos podemos usar una clase plantilla para parametrizar el receptor de la orden. Definiremos una subclase plantilla OrdenSimple para dichas órdenes. OrdenSimple es parametrizada con el tipo del Receptor y mantiene un enlace entre un objeto receptor y una acción almacenada como un puntero a una función miembro.

```
template <class Receptor>
class OrdenSimple : public Orden {
public:
     typedef void (Receptor::* Accion)();

     OrdenSimple(Receptor* r, Accion a) :
          _receptor(r), _accion(a) { }

     virtual void Ejecutar));
private:
```

```
        Accion _accion;
        Receptor* _receptor;
};
```

El constructor almacena el receptor y la acción las correspondientes-variables de instancia. Ejecutar simplemente aplica la acción al receptor.

```
template <class Receptor>
void OrdenSimple<Receptor>::Ejecutar () {
        (_receptor->*_accion)();
}
```

Para crear una orden que llame a Acción sobre una instancia de la clase MiClase, basta con que el cliente escriba

```
MiClase* receptor = new MiClase;
// …
Orden* unaOrden =
        new         OrdenSimple<MiClase>(receptor,
&MiClase::Accion);
// …
unaOrden->Ejecutar();
```

Hay que tener en cuenta que esta solución sólo sirve para órdenes simples. Otras órdenes más complejas que no sólo deban tratar con sus receptores sino también con argumentos y con información de estado para deshacer, necesitan una subclase de Orden.

Una OrdenMacro gestiona una secuencia de órdenes y proporciona operaciones para añadir y eliminar subórdenes. No se necesita un receptor explícito, ya que las subórdenes ya definen su receptor.

```
class OrdenMacro : public Orden {
public:
        OrdenMacro(); virtual ~OrdenMacro();

        virtual void Anadir(Orden*);
```

```
        virtual void Eliminar(Orden*);

        virtual void Ejecutar();
    private:
        Lista<Orden*>* _ordenes;
    };
```

Lo fundamental de OrdenMacro es su función miembro Ejecutar. Ésta recorre todas las subórdenes y llama a Ejecutar sobre cada una de ellas.

```
    void OrdenMacro::Ejecutar () {
        IteradorLista<Orden*> i(_ordenes);

        for   (i.Primero();   !i.HaTerminado();
    i.Siguiente()) {
            Orden* o = i.ElementoActual();
            o->Ejecutar();
        }
    }
```

Nótese que si OrdenMacro debe implementar una operación Deshacer, entonces sus subórdenes deben deshacerse en orden *inverso* con respecto a la implementación de Ejecutar.

Por último, OrdenMacro debe proporcionar operaciones para gestionar sus subórdenes, OrdenMacro es también responsable de borrar sus subórdenes.

```
    void OrdenMacro::Anadir (Orden* o) {
        _ordenes->Insertar(c);
    }
    void OrdenMacro::Eliminar (Orden* o) {
        _ordenes->Eliminar(c);
    }
```

USOS CONOCIDOS

Tal vez el primer ejemplo de patrón Command sea el que apareció en un artículo de Lieberman [Lie85], MacApp [App89] popularizó

la noción de órdenes para implementar operaciones que podían deshacerse. ET++ [WGM88], Interviews [LCI+92] y Unidraw [VL90] también definen clases que siguen el patrón Command. Interviews define una clase abstracta Action que proporciona la funcionalidad de una orden. También define una plantilla ActionCallback, parametrizada con un método de acción, que puede crear instancias de subclases de órdenes automáticamente.

La biblioteca de clases THINK [Sym93b] también usa órdenes para permitir acciones que se pueden deshacer. Las órdenes en THINK se denominan "Tasks" (tareas). Los objetos tarea se pasan a una Cadena de Responsabilidad (205), donde son consumidos.

Los objetos de órdenes de Unidraw son únicos en el sentido de que pueden comportarse como mensajes. Una orden de Unidraw puede enviarse a otro objeto para su interpretación, y el resultado de la interpretación varía con el objeto receptor. Más aún, el receptor puede delegar la interpretación a otro objeto, normalmente al padre en una estructura más grande, como en una Cadena de Responsabilidad. Así pues, el receptor de una orden de Unidraw se calcula, no se almacena. El mecanismo de interpretación de Unidraw depende de la información de tipos en tiempo de ejecución.

Coplien describe cómo implementar ***functors***, objetos que son funciones, en C++ [Cop92], Logra un grado de transparencia en su utilización sobrecargando el operador de llamada a función (operator()). El patrón Command es diferente; se centra en mantener un *enlace entre* un receptor y una función (es decir, una acción), no en mantener una función.

PATRONES RELACIONADOS

Se puede usar el patrón Composite (151) para implementar OrdenMacro.

Un Memento (261) puede mantener el estado que necesitan las órdenes para anular sus efectos. Una orden que debe ser copiada antes de ser guardada en el historial funciona como un Prototipo (109).

INTERPRETER (Intérprete)

PROPÓSITO

Dado un lenguaje, define una representación de su gramática junto con un intérprete que usa dicha representación para interpretar sentencias del lenguaje.

MOTIVACIÓN

Si hay un tipo de problemas que ocurren con cierta frecuencia, puede valer la pena expresar las apariciones de ese problema como instrucciones de un lenguaje simple. A continuación puede construirse un intérprete que resuelva el problema interpretando dichas instrucciones.

Por ejemplo, buscar cadenas que concuerden con un patrón es uno de estos problemas recurrentes. Las expresiones regulares son un lenguaje estándar para especificar patrones de cadenas. En vez de construir algoritmos personalizados que comparen cada patrón con diferentes cadenas, podríamos tener algoritmos de búsqueda que interpretasen una expresión regular que especifica el conjunto de cadenas a buscar.

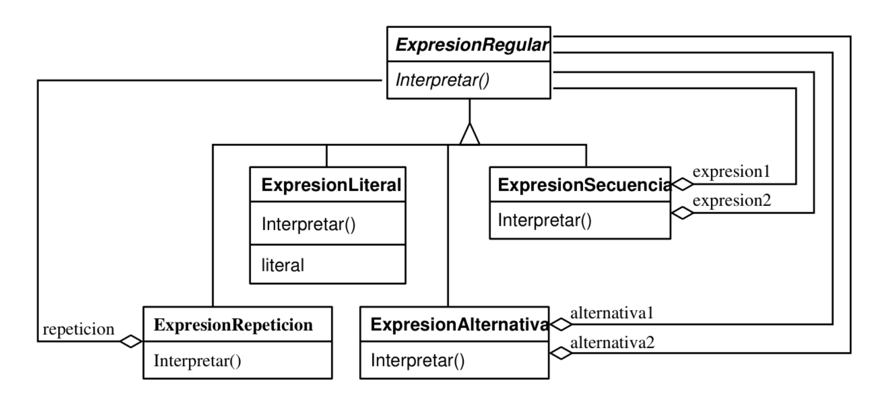
El patrón Interpreter describe cómo definir una gramática para lenguajes simples, cómo representar instrucciones de ese lenguaje y cómo interpretar esas instrucciones. En nuestro ejemplo, el patrón describe cómo definir una gramática para expresiones regulares, cómo representar una expresión regular concreta y cómo interpretar dicha expresión regular.

Supongamos que las expresiones regulares se definen mediante la siguiente gramática:

```
expresion  ::=  literal  ¦  alternativa  ¦
secuencia ¦ repeticion ¦
```

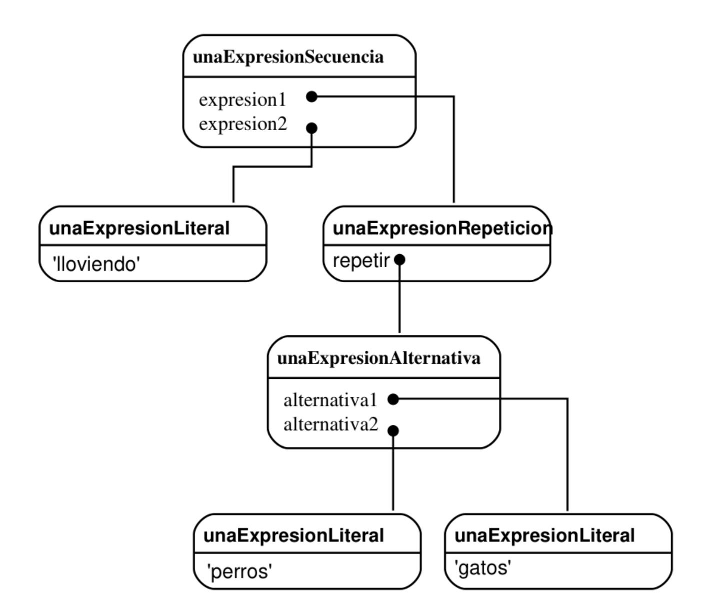
```
                    '(' expresion ')'
alternativa ::= expresion '¦' expresion
secuencia ::= expresion '&' expresion
repeticion ::= expresion '*'
literal ::= 'a' ¦ 'b' ¦ 'c' ¦ … { 'a' ¦ 'b' ¦
'c' ¦ … }*
```

El símbolo inicial es expresión, y literal es un símbolo terminal que define palabras.



El patrón Interpreter usa una clase para representar cada regla de la gramática. Los símbolos del lado derecho de la regla son variables de instancia de dichas clases. La gramática de más arriba se representa por cinco clases: una clase abstracta ExpresionRegular y sus cuatro subclases ExpresionLiteral, ExpresionAlternativa, ExpresionSecuencia y ExpresionRepeticion. Las últimas tres clases definen variables que contienen subexpresiones.

Cada expresión regular definida por esta gramática se representa por un árbol sintáctico abstracto formado por instancias de estas clases. Por ejemplo, el árbol sintáctico abstracto



representa la expresión regular

```
lloviendo 4 (perros ¦ gatos) *
```

Podemos crear un intérprete para estas expresiones regulares definiendo la operación Interpretaren cada subclase de ExpresionRegular. Interpretar toma como argumento el contexto en el cual se interpreta la expresión. El contexto contiene la cadena de entrada e información acerca de qué parte de ella se ha reconocido hasta el momento. Cada subclase de ExpresionRegular implementa Interpretar para reconocer la siguiente parte de la cadena de entrada en función del contexto actual. Por ejemplo,

- ExpresionLiteral comprobará si la entrada coincide con el literal que ella define,
- ExpresionAltemativa comprobará si la entrada coincide con alguna de sus alternativas,
- ExpresionRepeticion comprobará si la entrada tiene múltiples copias de la expresión repetida,

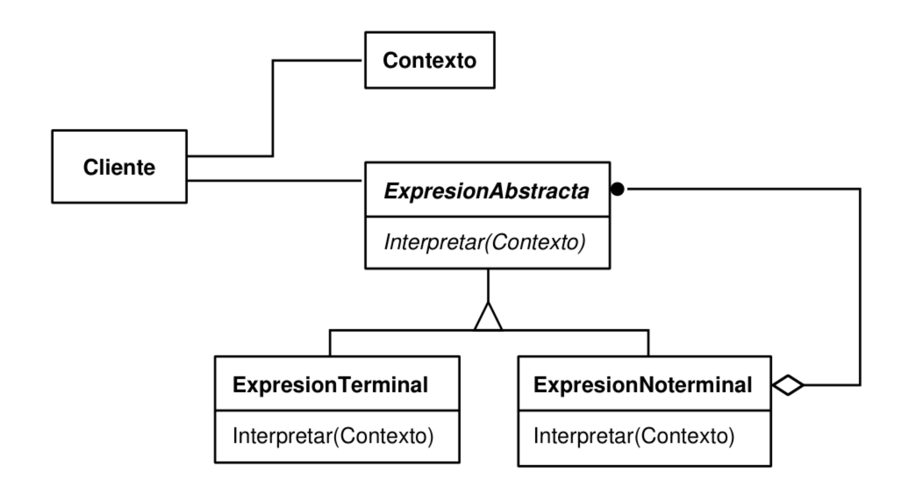
y así sucesivamente.

APLICABILIDAD

Úsese el patrón Interpreter cuando hay un lenguaje que interpretar y se pueden representar las sentencias del lenguaje como árboles de sintáctico 1 abstractos. El patrón Interpreter funciona mejor cuando

- la gramática es simple. Para gramáticas complejas, la jerarquía de clases de la gramática se vuelve grande e inmanejable. Herramientas como los generadores de analizadores sintácticos constituyen una alternativa mejor en estos casos. Éstas pueden interpretar expresiones sin necesidad de construir árboles sintácticos abstractos, lo que puede ahorrar espacio y, posiblemente, tiempo.
- la eficiencia no es una preocupación crítica. Los intérpretes más eficientes normalmente *no* se implementan interpretando árboles de análisis sintáctico directamente, sino que primero los traducen a algún otro formato. Por ejemplo, las expresiones regulares suelen transformarse en máquinas de estados. Pero incluso en ese caso, el *traductor* puede implementarse con el patrón Interpreter, de modo que éste sigue siendo aplicable.

ESTRUCTURA



PARTICIPANTES

- **ExpresionAbstracta** (ExpresionRegular)
  - declara una operación abstracta Interpretar que es común a todos los nodos del árbol de sintaxis abstracto.
- **ExpresionTerminal** (ExpresionLiteral)
  - implementa una operación Interpretar asociada con los símbolos terminales de la gramática.
  - se necesita una instancia de esta clase para cada símbolo terminal de una sentencia.
- **ExpresionNoTerminal** (ExpresionAltemativa, ExpresionRepeticion, ExpresionSecuencia)
  - por cada regla de la gramática $R ::=$ R_1, R_2 ... Rn debe haber una de estas clases.
  - mantiene variables de instancia de tipo ExpresionAbstracta para cada uno de los símbolos de R_1 a R_n.
  - implementa una operación Interpretar para lo símbolos no terminales de la gramática. Interpretar normalmente se llama a sí misma recursivamente sobre las variables que representan de R_1 a R_n
- **Contexto**
  - contiene información que es global al intérprete.
- **Cliente**
  - construye (o recibe) un árbol sintáctico abstracto que representa una determinada sentencia del lenguaje definido por la gramática. Este árbol sintáctico abstracto está formado por instancias de las clases ExpresionNoTerminal y ExpresionTerminal.
  - invoca a la operación Interpretar.

COLABORACIONES

- El cliente construye (o recibe) la sentencia como un árbol sintáctico abstracto formado por instancias de ExpresionNoTerminal y ExpresionTerminal. A continuación el cliente inicializa el contexto e invoca a la operación Interpretar.
- Cada nodo ExpresionNoTerminal define Interpretar en términos del Interpretar de cada subexpresión. La operación Interpretar de cada ExpresionTerminal define el caso base de la recursión.
- Las operaciones Interpretar de cada nodo usan el contexto para almacenar y acceder al estado del intérprete.

CONSECUENCIAS

El patrón Interpreter tiene las siguientes ventajas e inconvenientes:

1. *Es fácil cambiar y ampliar la gramática*. Puesto que el patrón usa clases para representar las reglas de la gramática, se puede usar la herencia para cambiar o extender ésta. Se puede modificar incrementalmente las expresiones existentes, y se pueden definir otras nuevas como variaciones de las antiguas.
2. *También resulta fácil implementar la gramática*. Las clases que definen los nodos del árbol sintáctico abstracto tienen implementaciones similares. Estas clases son fáciles de escribir, y muchas veces se pueden generar automáticamente con un compilador o un generador de analizadores sintácticos.
3. *Las gramáticas complejas son difíciles de mantener*. El patrón Interpreter define al menos una clase para cada regla de la gramática (las reglas que se hayan definido usando BNF pueden necesitar varias clases). De ahí que las gramáticas que contienen muchas reglas pueden ser difíciles de controlar y mantener. Se pueden aplicar otros patrones de diseño para mitigar el problema (véase la sección de Implementación). Pero cuando la gramática es muy compleja son más

adecuadas otras técnicas como los generadores de analizadores sintácticos o de compiladores.

4. *Añadir nuevos modos de interpretar expresiones*. El patrón Interpreter facilita evaluar una expresión de una manera distinta. Por ejemplo, podríamos permitir imprimir con formato una expresión o realizar una comprobación de tipos en ella definiendo una nueva operación en las clases de las expresiones. Si vamos a seguir añadiendo nuevos modos de interpretar una expresión, deberíamos considerar la utilización del patrón Visitor (305) para evitar cambiar las clases de la gramática.

IMPLEMENTACIÓN

Los patrones Interpreter y Composite (151) comparten muchos detalles de implementación. Las siguientes cuestiones son específicas del Interpreten

1. *Crear un árbol sintáctico abstracto*. El patrón Interpreter no explica cómo *crear* un árbol sintáctico abstracto. En otras palabras, no se encarga del análisis sintáctico. El árbol sintáctico abstracto puede ser creado mediante un analizador sintáctico dirigido por una tabla, mediante un analizador sintáctico (normalmente recursivo descendente) hecho a mano, o directamente por el cliente.

2. *Definir la operación Interpretar*. No tenemos por qué definir la operación Interpretar en las clases de la expresión. Si se van a crear nuevos intérpretes es mejor usar el patrón Visitor (305) para poner Interpretar en un objeto "visitante" aparte. Por ejemplo, una gramática de un lenguaje de programación tendrá muchas operaciones sobre los árboles sintácticos abstractos, tales como la comprobación de tipos, la optimización, la generación de código, etcétera. Será mejor usar un visitante para evitar definir estas operaciones en cada clase de la gramática.

3. *Compartir símbolos terminales mediante el patrón Flyweight*. Para las gramáticas cuyas sentencias contienen

muchas repeticiones de un mismo símbolo terminal puede ser beneficioso compartir una única copia de dicho símbolo. Las gramáticas para programas de computadora son buenos ejemplos —cada variable de un programa aparecerá en muchos sitios a lo largo del código—. En el ejemplo de la sección de Motivación, una sentencia puede tener el símbolo terminal perro (modelado mediante la clase ExpresionLiteral) repetido muchas veces.

Los nodos terminales generalmente no guardan información sobre su posición en el árbol sintáctico abstracto. Los nodos padre les pasan cualquier contexto que pudieran necesitar durante la interpretación. Por tanto aquí se da una distinción entre estado compartido (intrínseco) y estado recibido (extrínseco), por lo que es aplicable el patrón Flyweight (179).

Por ejemplo, cada instancia de ExpresionLiteral para perro recibe un contexto que contiene la subcadena reconocida hasta ese instante. Y cada una de estas instancias de ExpresionLiteral hace lo mismo en su operación Interpretar —comprueba si la parte que sigue de la entrada contiene un perro—, sin importar en qué posición del árbol aparezca la instancia.

CÓDIGO DE EJEMPLO

A continuación se mostrarán dos ejemplos. El primero de ellos es un código completo en Smalltalk para comprobar si una secuencia satisface una expresión regular. El segundo es un programa en C++ para evaluar expresiones booleanas.

El reconocedor de expresiones regulares comprueba si una cadena pertenece al lenguaje definido por la expresión regular. La expresión regular está definida por la siguiente gramática:

```
expresion   ::=   literal   ¦   alternativa   ¦
secuencia ¦ repeticion ¦
                  '(' expresion ')'
alternativa ::= expresion '¦' expresion
secuencia ::= expresion '&' expresion
```

```
repeticion ::= expresion 'repetir'
literal ::= 'a' ¦ 'b' ¦ 'c' ¦ … { 'a' ¦ 'b' ¦
'c' ¦ … }*
```

Esta gramática es una ligera modificación de la del ejemplo de la sección de Motivación. Hemos cambiado un poco la sintaxis concreta de las expresiones regulares, ya que el símbolo “*” no puede ser una operación postfija en Smalltalk. Por ese motivo usaremos repetir en su lugar. Por ejemplo, la expresión regular

```
(('perro ' ¦ 'gato ')
repetir & 'tiempo')
```

reconoce la cadena de entrada 'perro perro gato tiempo'.

Para implementar el reconocedor, definimos las cinco clases descritas anteriormente. La clase ExpresionSecuencia tiene las variables de instancia expresionl y expresion2 para sus hijos en el árbol sintáctico abstracto. ExpresionAltemativa guarda sus alternativas en las variables de instancia alternativa1 y alternativa2, mientras que ExpresionRepeticion guarda en su variable de instancia repetición la expresión que se repite. ExpresionLiteral tiene una variable de instancia componentes que contiene una lista de objetos (probablemente caracteres). Éstos representan la cadena literal con la que debe coincidir la secuencia de entrada.

La operación reconocer: implementa un intérprete de expresiones regulares. Cada una de las clases que definen el árbol sintáctico abstracto implementa esta operación, que recibe en el parámetro estadoEntrada el estado actual del proceso de análisis, tras haber le(do parte de la cadena de entrada.

Dicho estado actual viene dado por una serie de flujos de entrada que representan todas las entradas que podría haber aceptado hasta el momento la expresión regular. (Esto equivale más o menos a guardar todos los estados en los que estaría el autómata de estados finitos equivalente tras haber reconocido la cadena de entrada hasta este punto.)

El estado actual es de vital importancia para la operación repetir. Por ejemplo, si la expresión regular fuera

```
'a' repetir
```

entonces el intérprete podría reconocer "a", "aa", "aaa" y así sucesivamente. Si fuera

```
'a' repetir & 'bc'
```

entonces podría reconocer "abc", "aabc", "aaabc", etcétera. Pero si la expresión regular fuese

```
'a' repetir & 'abc'
```

entonces comparar la entrada "aabc" con la subexpresión "'a' repetir" daría lugar a dos flujos de entrada, uno en el que se reconociera un carácter de la entrada y otro en el que se reconocieran dos caracteres. Sólo el flujo que ha aceptado un carácter reconocerá el "abc" restante.

Pensemos ahora en las definiciones de reconocer: para cada clase que define una expresión regular. La definición de esta operación para ExpresionSecuencia reconoce cada una de sus subexpresiones secuencialmente. Normalmente eliminará flujos de entrada de su estadoEntrada.

```
reconocer: estadoEntrada
     ^ expresion2 reconocer: (expresion1
reconocer: estadoEntrada).
```

Una ExpresionAlternativa devolverá un estado que consiste en la unión de estados de cada alternativa. La definición de reconocer: para ExpresionAlternativa es

```
reconocer: estadoEntrada
     | estadoFinal |
     estadoFinal := alternativa) reconocer:
```

```
estadoEntrada.
     estadoFinal    addAll:    (alternativa2
reconocer: estadoEntrada).
     ^ estadoFinal
```

La operación reconocer: de ExpresionRepeticion trata de encontrar tantos estados que pueda reconocer como sea posible:

```
reconocer: estadoEntrada
     ¦ unEstado estadoFinal ¦
     unEstado := estadoEntrada.
     estadoFinal := estadoEntrada copy.
     [unEstado isEmpty]
          whileFalse:
               [unEstado    :=    repeticion
reconocer: unEstado.
               estadoFinal addAll: unEstado].
          ^ estadoFinal
```

Su estado de salida normalmente consiste en más estados que su estado de entrada, ya que ExpresionRepeticion puede reconocer uno, dos o muchas apariciones de repetición en el estado de entrada. Los estados de salida representan todas estas posibilidades, permitiendo a los siguientes elementos de la expresión regular decidir qué estado es el correcto.

Por último, la definición de reconocer: para ExpresionLiteral trata de reconocer sus componentes para cada posible flujo de entrada. Sólo mantiene aquellos flujos de entrada que concuerdan con la expresión:

```
reconocer: estadoEntrada
     ¦ estadoFinal tStream ¦
     estadoFinal : = Set new.
     estadoEntrada
          do:
               [:stream  ¦  tStream  :=  stream
copy.
                    (tStream nextAvailable:
                         components size
                    ) * components
```

```
                            ifTrue: [estadoFinal
add: tStream]
              ].
          ^ estadoFinal
```

El mensaje siguienteDisponible: hace que avance el flujo de entrada. Ésta es la única operación reconocer: que hace avanzar al flujo. Nótese cómo el estado que devuelve contiene una copia del flujo de entrada, garantizando así que reconocer un literal nunca cambie el flujo de entrada. Esto es importante porque todas las alternativas de una ExpresionAltemativa deberían ver copias idénticas del flujo de entrada.

Ahora que hemos definido las clases de las que se compone un árbol sintáctico abstracto, estamos en condiciones de describir cómo construirlo. En vez de escribir un analizador sintáctico de expresiones regulares, definiremos algunas operaciones en las clases ExpresionRegular de forma que evaluar una expresión en Smalltalk produzca un árbol sintáctico abstracto para la correspondiente expresión regular. Eso nos permite usar el compilador incorporado de Smalltalk como si fuese un analizador sintáctico de expresiones regulares.

Para construir el árbol sintáctico abstracto necesitaremos definir "¦", "repetir" y "&" como operaciones de ExpresionRegular. Estas operaciones se definen en la clase ExpresionRegular como sigue:

```
& unNodo
     ^ ExpresionSecuencia new
          expresionl: self expresion2: unNodo
comoExpReg

repetir
     ^  ExpresionRepeticion  new  repeticion:
self

¦ unNodo
     ^ ExpresionAltemativa new
     alternatival:  self  alternativa2:  unNodo
comoExpReg

comoExpReg
     ^ self
```

La operación comoExpReg convertirá literales en objetos ExpresionRegular. Estas operaciones se definen en la clase String:

```
& unNodo
     ^ ExpresionSecuencia new
          expresionl:     self     comoExpReg
expresion2: unNodo comoExpReg

repetir
     ^  ExpresionRepeticion  new  repeticion:
self

¦ unNodo
     ^ ExpresionAltemativa new
          alternatival:    self    comoExpReg
alternativa2: unNodo comoExpReg

comoExpReg
     ^ ExpresionLiteral new componentes: self
```

Si definiéramos estas operaciones más arriba en la jerarquía de clases (SequenceableCollection en Smalltalk-80, IndexedCollection en Smalltalk/V), entonces también estarían definidas para clases como Array y OrderedCollection. Esto haría que las expresiones regulares reconociesen secuencias de objetos de cualquier tipo.

El segundo ejemplo es un sistema para manipular y evaluar expresiones booleanas, implementado en C++. Los símbolos terminales del lenguaje son variables de tipo Boolean, es decir, las constantes true y false. Los símbolos no terminales representan expresiones que contienen los operadores and, or, y not. La gramática se define como sigue [49]:

```
ExpBooleana  ::=  ExpVariable  ¦  Constante  ¦
ExpOr ¦ ExpAnd ¦ ExpNot ¦
                 '(' ExpBooleana ')'
ExpAnd ::= ExpBooleana 'and' ExpBooleana
ExpOr ::= ExpBooleana 'or' ExpBooleana
ExpNot ::= 'not' ExpBooleana
Constante ::= 'true' ¦ 'false'
ExpVariable ::= 'A' ¦ 'B' ¦ … ¦ 'X' ¦ 'Y' ¦
'Z'
```

Definiremos dos operaciones para las expresiones booleanas. La primera, Evaluar, evaluaría una expresión Booleana en un contexto que asigna un valor verdadero o falso a cada variable. La segunda operación, Sustituir, produce una nueva expresión booleana al sustituir una variable por una expresión. Sustituir muestra cómo se puede usar el patrón Interpreter para algo más que simplemente evaluar expresiones. En este caso, para manipular la propia expresión.

A continuación se detallarán las clases ExpBooleana, ExpVariable y ExpAnd. Las clases ExpOr y ExpNot son parecidas a ExpAnd. La clase Constante representa las constantes lógicas.

ExpBooleana define la interfaz para todas las clases que definen una expresión booleana:

```
class ExpBooleana {
public:
     ExpBooleana();
     virtual ~ExpBooleana();

     virtual bool Evaluar(Contexto&) = 0;
     virtual   ExpBooleana*   Sustituir(const
char*, ExpBooleana&) = 0;
     virtual ExpBooleana* Copiar() const = 0;
};
```

La clase contexto define una correspondencia entre variables y valores booleanos, los cuales se representan mediante las constantes de C++ true y false. Contexto tiene la siguiente interfaz:

```
class Contexto {
public:
     bool Buscar(const char*) const;
     void Asignar(ExpVariable*, bool);
};
```

Una ExpVariable representa una variable con nombre:

```
class ExpVariable : public ExpBooleana {
```

```
public:
     ExpVariable(const char*);
     virtual ~ExpVariable();

     virtual bool Evaluar(Contexto&);
     virtual   ExpBooleana*   Sustituir(const
char*, ExpBooleana&);
     virtual ExpBooleana* Copiar() const;
private:
     char* _nombre;
};
```

El constructor recibe como parámetro el nombre de la variable:

```
ExpVariable::ExpVariable (const char* nombre)
{
     _nombre = strdup(nombre);
}
```

Evaluar una variable devuelve su valor en el contexto actual.

```
bool      ExpVariable::Evaluar      (Contexto&
unContexto) {
     return unContexto.Buscar(_nombre);
}
```

Copiar una variable devuelve una nueva ExpVariable:

```
ExpBooleana* ExpVariable::Copiar () const {
     return new ExpVariable(_nombre);
}
```

Para sustituir una variable por una expresión hemos de comprobar si la variable tiene el mismo nombre que la que se recibe como parámetro:

```
ExpBooleana* ExpVariable::Sustituir (
```

```
    const char* nombre, ExpBooleana& exp
) {
    if (strcmp(nombre, _nombre) == 0) {
        return exp.Copiar();
    } else {
        return new ExpVariable(_nombre);
    }
}
```

Por otro lado, ExpAnd representa una expresión resultado de unir dos expresiones booleanas por la operación “y” lógica.

```
class ExpAnd : public ExpBooleana {
public:
    ExpAnd(ExpBooleana*, ExpBooleana*);
    virtual ~ ExpAnd();

    virtual bool Evaluar(Contexto&);
    virtual  ExpBooleana*  Sustituir(const
char*, ExpBooleana&);
    virtual ExpBooleana* Copiar() const;
private:
    ExpBooleana* _operando1;
    ExpBooleana* _operando2;
};
ExpAnd::ExpAnd      (ExpBooleana*       op1,
ExpBooleana* op2) {
    _operando1 = op1;
    operando2 = op2;
}
```

Evaluar una ExpAnd consiste en evaluar sus operandos y devolver el “y” lógico de los resultados.

```
bool ExpAnd:: Evaluar (Contexto* unContexto)
{
    return
        _operando1->Evaluar(unContexto) &&
        _operando2->Evaluar(unContexto);
}
```

Una ExpAnd implementa Copiar y Sustituir haciendo llamadas recursivas sobre sus operandos:

```
ExpBooleana* ExpAnd::Copiar () const {
     return
          new    ExpAnd(_operando1->Copiar(),
operando2->Copiar());
}

ExpBooleana*  ExpAnd::Sustituir  (const  char*
nombre, ExpBooleana& exp) {
     return
          new ExpAnd(
               _operando1->Sustituir(nombre,
exp),
               _operando2->Sustituir(nombre,
exp)
          );
}
```

Ahora podemos definir la expresión booleana

```
(true and x) or (y and (not x))
```

y evaluarla asignando true o false a las variables x e y:

```
ExpBooleana* expresion;
Contexto contexto;

ExpVariable* x = new ExpVariable("X");
ExpVariable* y = new ExpVariable("Y*);

expresión = new ExpOr(
     new ExpAnd(new Constante(true), x),
     new ExpAnd(y, new ExpNot(x))
);

contexto.Asignar(x, false);
contexto.Asignar(y, true);

bool resultado = expresion->Evaluar(context);
```

La expresión se evalúa a true para estos valores de x e y. Podemos evaluar la expresión dándoles diferentes valores a las variables simplemente cambiando el contexto.

Por último, podemos sustituir la variable y con una nueva expresión y luego volver a evaluarla:

```
ExpVariable* z = new ExpVariable("Z");
ExpNot not_z(z);

ExpBooleana*    sustitucion    =    expresion-
>Sustituir("Y", not_z);

contexto.Asignar(z, true);

resultado = sustitucion->Evaluar(context);
```

Este ejemplo ilustra un aspecto importante del patrón Interpreter: hay muchos tipos de operaciones que pueden "interpretar" una sentencia. De las tres operaciones definidas para ExpBooleana, Evaluar es la que más se ajusta a nuestra idea de lo que debería hacer un intérprete —interpretar un programa o expresión y devolver un resultado simple—.

No obstante, Sustituir también puede verse como un intérprete. Es un intérprete cuyo contexto es el nombre de la variable que está siendo sustituida junto con la expresión que la sustituye, y cuyo resultado es una nueva expresión. Incluso podría pensarse en Copiar como un intérprete con un contexto vacío. Puede parecer un poco extraño considerar a Sustituir y Copiar como intérpretes, dado que éstas son operaciones básicas sobre árboles. Los ejemplos del patrón Visitor (305) muestran cómo estas tres operaciones pueden refactorizarse en un “intérprete” visitante aparte, revelando así una profunda similitud.

El patrón Interpreter es más que una simple operación distribuida sobre una jerarquía de clases que usa el patrón Composite (151). Si consideramos a Evaluar como un intérprete es porque pensamos en la jerarquía de clases de ExpBooleana como la representación de un lenguaje. Supuesta una jerarquía de clases similar para representar el ensamblaje de partes de automóviles, no es probable que hubiéramos considerado intérpretes a operaciones

como Paso y Copiar, aunque estén distribuidas sobre una jerarquía de clases que usa el patrón Composite —no pensamos en las partes de un automóvil como un lenguaje—. Es una cuestión de perspectiva; si empezásemos a publicar gramáticas de parles de automóviles, entonces se podría considerara las operaciones sobre esas partes como formas de interpretar el lenguaje.

USOS CONOCIDOS

El patrón Interpreter está muy extendido en los compiladores implementados con lenguajes orientados a objetos, como los compiladores de Smalltalk. SPECTalk usa este patrón para interpretar descripciones de formatos de ficheros de entrada [Sza92], El toolkit de resolución de problemas QOCA lo usa para evaluar los problemas [HHMV92],

Concebido en su forma más general (es decir, como una operación distribuida sobre una jerarquía de clases basada en el patrón Composite), casi cualquier uso del patrón Composite también contendrá el patrón Interpreter. Pero este patrón debería reservarse para aquellos casos en los que tiene sentido pensar en la jerarquía de clases como la definición de un lenguaje.

PATRONES RELACIONADOS

Composite (151); el árbol sintáctico abstracto es una instancia del patrón Composite.

El patrón Flyweight (179) muestra cómo compartir símbolos terminales dentro del árbol sintáctico abstracto.

Iterator (237): el intérprete puede usar un Iterador para recorrer la estructura.

Puede usarse el patrón Visitor (305) para mantener el comportamiento de cada nodo del árbol sintáctico abstracto en una clase.

ITERATOR (Iterador)

PROPÓSITO

Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

TAMBIÉN CONOCIDO COMO

Cursor

MOTIVACIÓN

Un objeto agregado, como por ejemplo una lista, debería darnos una forma de acceder a sus elementos sin exponer su estructura interna. Más aún, tal vez queramos recorrer la lista de diferentes formas, dependiendo de lo que queramos realizar. Pero probablemente no querremos plagar la interfaz de Lista con operaciones para diferentes recorridos, incluso en el caso de que pudiéramos prever cuáles se van a necesitar. Por otro lado, también puede necesitarse hacer más de un recorrido simultáneamente sobre la misma lista.

El patrón Iterator nos permite hacer todo esto. La idea clave de este patrón es tomar la responsabilidad de acceder y recorrer el objeto lista y poner dicha responsabilidad en un objeto iterador. La clase Iterador define una interfaz para acceder a los elementos de la lista. Un objeto iterador es el responsable de saber cuál es el elemento actual; es decir, sabe qué elementos ya han sido recorridos.

Por ejemplo, una clase Lista pediría un IteradorLista con la siguiente relación entre ambos:

Lista	lista	IteradorLista
Contar() Insertar(Elemento) Eliminar(Elemento) ...		Primero() Siguiente() HaTerminado() ElementoActual()
		indice

Antes de que pueda crearse una instancia de IteradorLista debemos proporcionarle la Lista a recorrer. Una vez que tenemos la instancia de IteradorLista, podemos acceder secuencialmente a los elementos de la lista. La operación ElementoActual devuelve el elemento actual de la lista; Primero inicializa el elemento actual al primer elemento; Siguiente hace avanzar el elemento actual al siguiente elemento; y HaTerminado comprueba si se ha avanzado más allá del último elemento, es decir, si se ha finalizado el recorrido.

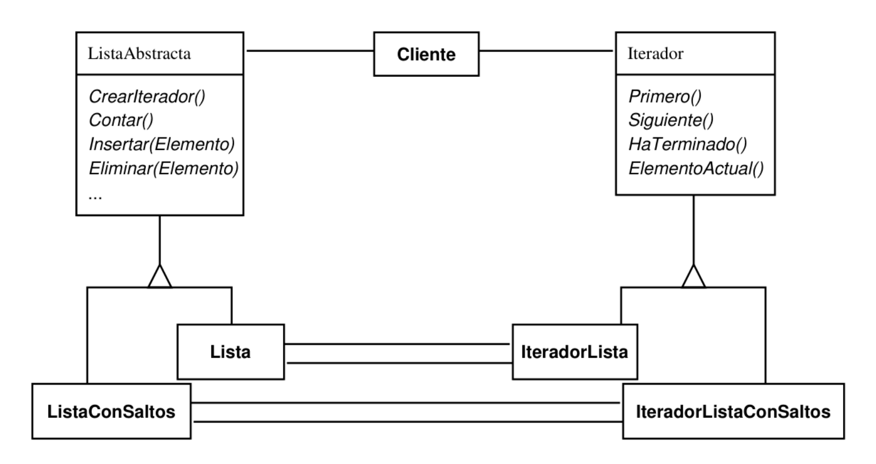
Separar el mecanismo de recorrido del objeto Lista nos permite definir iteradores con diferentes políticas de recorrido sin necesidad de enumerarlos en la interfaz de Lista. Por ejemplo, IteradorListaConFiltro podría proporcionar acceso sólo a aquellos elementos que satisfagan las normas de filtrado.

Nótese que el iterador y la lista están acoplados, y que el cliente debe saber que lo que se está recorriendo es una *lista* y no otra estructura agregada. Por tanto, el cliente se ajusta a una determinada estructura agregada. Sería mejor que pudiésemos cambiar la clase agregada sin cambiar el código cliente. Podemos hacer esto por generalización del concepto de iterador para permitir la **Iteración polimórfica**.

Como ejemplo, supongamos que ya tenemos una implementación de una lista ListaConSaltos. Una lista con saltos (*skiplist*) [Pug90] es una estructura de datos con características similares a los árboles equilibrados. Nos gustaría poder escribir código que funcionase tanto para objetos Lista como para objetos ListaConSaltos.

Definimos una clase ListaAbstracta que proporciona una interfaz común para manipular listas. Igualmente, necesitamos una clase abstracta Iterador que defina una interfaz de iteración común. Entonces podemos definir las subclases concretas de Iterador para

las diferentes implementaciones de listas. Como resultado, el mecanismo de iteración se vuelve independiente de las clases agregadas concretas.



Nos queda el problema de cómo crear el iterador. Puesto que queremos escribir código que sea independiente dé las subclases concretas de Lista, no podemos crear simplemente una instancia de una clase determinada. En vez de eso, haremos que los objetos lista sean responsables de crear sus correspondientes iteradores. Esto requiere una operación como CrearIterador, mediante la cual los clientes soliciten un objeto iterador.

CrearIterador es un ejemplo de método de fabricación (*véase* el patrón Factory Method (99)). Aquí Jo usamos para permitir que un cliente le pida a un objeto lista el iterador apropiado. El enfoque seguido con el patrón Factory Method da lugar a dos jerarquías de clases, una para las listas y otra para los iteradores. El método de fabricación CrearIterador "conecta" las dos jerarquías.

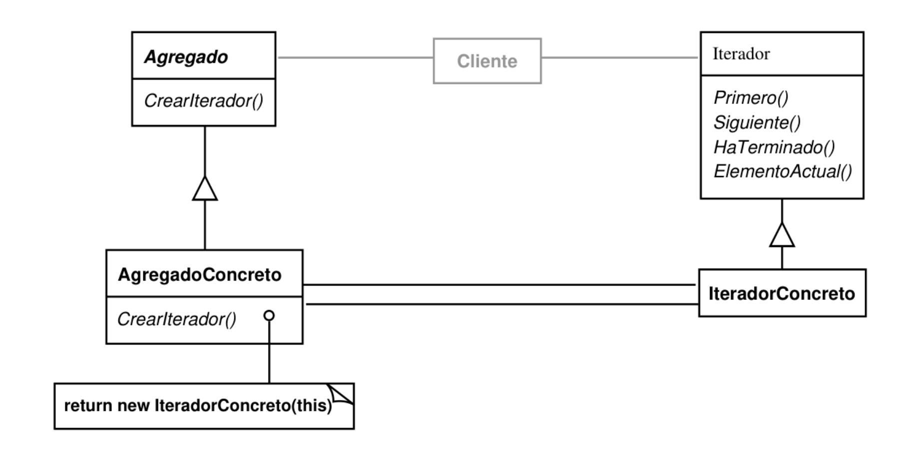
APLICABILIDAD

Úsese el patrón Iterador

- para acceder al contenido de un objeto agregado sin exponer su representación interna.
- para permitir varios recorridos sobre objetos agregados.
- para proporcionar una interfaz uniforme para recorrer diferentes estructuras agregadas (es decir, para permitir la

iteración polimórfica).

ESTRUCTURA



PARTICIPANTES

- **Iterador**
  - define una interfaz para recorrer los elementos y acceder a ellos.
- **IteradorConcreto**
  - implementa la interfaz Iterador.
  - mantiene la posición actual en el recorrido del agregado.
- **Agregado**
  - define una interfaz para crear un objeto Iterador.
- **AgregadoConcreto**
  - implementa la interfaz de creación de Iterator para devolver una instancia del IteradorConcreto apropiado.

COLABORACIONES

- Un IteradorConcreto sabe cuál es el objeto actual del agregado y puede calcular el objeto siguiente en el recorrido.

CONSECUENCIAS

El patrón Iterator tiene tres consecuencias importantes:

1. *Permite variaciones en el recorrido de un agregado*. Los agregados complejos pueden recorrerse de muchas formas. Por ejemplo, la generación de código y la comprobación de tipos implican recorrer árboles de análisis sintáctico. La generación de código puede recorrer dicho árbol de análisis sintáctico en en-orden o en pre-orden. Los iteradores facilitan cambiar el algoritmo de recorrido: basta con sustituir la instancia de iterador por otra diferente. También se pueden definir subclases de Iterador pura permitir nuevos recorridos.
2. *Los iteradores simplifican la interfaz Agregado*. La interfaz de recorrido de Iterador elimina la necesidad de una interfaz parecida en Agregado, simplificando así la interfaz del agregado.
3. *Se puede hacer más de un recorrido a la vez sobre un agregado*. Un iterador mantiene su propio estado del recorrido. Por tanto, es posible estar realizando más de un recorrido al mismo tiempo.

IMPLEMENTACIÓN

El patrón Iterator tiene muchas variantes y alternativas de implementación. A continuación se muestran algunas de las más importantes. Los pros y los contras muchas veces dependen de las estructuras de control proporcionadas por el lenguaje. Algunos lenguajes (CLU [LG86], por ejemplo) llegan a incluir este patrón directamente.

1. ¿*Quién controla la iteración*? Una cuestión fundamental es decidir qué participante controla la iteración, si el iterador o el cliente que lo usa. Cuando es el cliente quien controla la iteración, el iterador se denomina **iterador externo**, y cuando es el iterador quien la controla, se dice que el iterador es un **iterador interno**[50]. Los clientes que usan un iterador externo deben avanzar en el recorrido y

pedirle explícitamente al iterador el siguiente elemento. En el caso contrario, el cliente maneja un iterador interno y éste aplica esa operación a cada elemento del agregado.

Los iteradores estemos son más flexibles que los internos. Por ejemplo, resulta fácil comparar dos colecciones para ver si son iguales usando un iterador externo, pero esto mismo es prácticamente imposible con iteradores internos. Los iteradores internos son especialmente débiles en un lenguaje como C++, que no proporciona funciones anónimas, cierres o reanudaciones como sí hacen Smalltalk y CLOS. Pero, por otro lado, los iteradores internos son más fáciles de usar, ya que definen la lógica de iteración por nosotros.

2. ¿*Quién define el algoritmo de recorrido*? El iterador no es el único lugar donde se puede definir el algoritmo de recorrido. El agregado podría definir el algoritmo de recorrido y usar el iterador para almacenar sólo el estado de la iteración. A este tipo de iterador lo denominamos cursor, ya que se limita a apuntar a la posición actual del agregado. Un cliente invocará a la operación Siguiente sobre el agregado con el cursor como parámetro, y la operación Siguiente cambiará e] estado del cursor [51].

   Si el iterador es el responsable del algoritmo de recorrido, entonces es fácil usar diferentes algoritmos de iteración sobre el mismo agregado, y también puede ser más fácil reutilizar el mismo algoritmo sobre diferentes agregados. Por otro lado, el algoritmo de recorrido puede necesitar acceder a las variables privadas del agregado. Si es así, poner el algoritmo de recorrido en el iterador violaría la encapsulación del agregado.

3. ¿*Cómo es de robusto el iterador*? Puede ser peligroso modificar un agregado mientras lo estamos recorriendo. Si se añaden o borran elementos del iterador podríamos acabar accediendo dos veces a un elemento o a uno que ya no existe. Una solución sencilla es copiar el agregado y recorrer la copia, pero eso es demasiado costoso como

para hacerlo siempre.

Un iterador robusto garantiza que las inserciones y borrados no interferirán con el recorrido, y lo hace sin copiar el agregado. Hay muchas formas de implementar iteradores robustos. La mayoría se basan en registrar el iterador con el agregado. Al insertar o borrar, el agregado ajusta el estado interno de los iteradores que ha producido, o mantiene información internamente para garantizar un recorrido apropiado.

Kofler proporciona una buena discusión acerca de cómo están implementados los iteradores robustos en ET++ [Kof93], Murray examina la implementación de iteradores robustos para la clase List de USL StandardComponents [Mur93].

4. *Operaciones adicionales de Iterador*. La interfaz mínima de Iterador consiste en las operaciones Primero, Siguiente, HaTerminado y ElementoActual[52]. Podrían ser útiles algunas operaciones adicionales. Por ejemplo, los agregados ordenados pueden tener una operación Anterior que posicione el iterador en el elemento anterior. Una operación IrA es útil para ordenar o indexar colecciones. IrA posiciona el iterador en un objeto que cumpla el criterio especificado.

5. *Usar iteradores polimórficos en* C++. Los iteradores polimórficos tienen un coste. Necesitan que el objeto iterador sea creado dinámicamente por un método de fabricación. Por tanto deberían usarse sólo cuando hay necesidad de polimorfismo. En caso contrario, es mejor usar iteradores concretos, que pueden crearse en la pila.

   Los iteradores polimórficos tienen otro inconveniente: el cliente es el responsable de borrarlos. Esto es propenso a errores, ya que es fácil olvidarse de liberar la memoria asignada a un iterador cuando se ha terminado de usarlo. Eso es especialmente probable cuando en una operación hay múltiples puntos de salida. Y si se lanza una

excepción, el objeto iterador nunca se liberará.

El patrón Proxy (191) proporciona un remedio para esto. Podemos usar un proxy en la memoria de pila como sustituto del iterador real. El proxy elimina el iterador en su destructor. Así, cuando el proxy alcanza el final de su ámbito, la memoria asignada al iterador real será liberada con él. El proxy garantiza una limpieza adecuada, incluso en el caso de que se produzcan excepciones. Ésta es una aplicación de la conocida técnica de C++ "asignación de recursos es inicialización" [ES90]. El Código de Ejemplo ofrece un ejemplo de esto.

6. *Los iteradores pueden tener un acceso restringido*. Podemos ver a un iterador como una extensión del agregado que lo crea. El iterador y el agregado están fuertemente acoplados. En C++ se puede expresar esta estrecha relación haciendo que el iterador sea una clase friend de su agregado. En ese caso ya no es necesario definir operaciones en el agregado cuyo único propósito es permitir a los iteradores implementar el recorrido de manera eficiente.

   Sin embargo, este acceso restringido puede dificultar la definición de nuevos recorridos, ya que requerirá cambiar la interfaz del agregado para añadir otra clase amiga. Para evitar este problema, la clase Iterador puede incluir operaciones de tipo protected para acceder a los miembros importantes, pero que no están disponibles públicamente, del agregado. Las subclases de Iterador (y *sólo* ellas) pueden usar estas operaciones protegidas para obtener un acceso restringido al agregado.

7. *Iteradores en lugar de compuestos*. Los iteradores externos pueden ser difíciles de implementar sobre estructuras agregadas recursivas, como las del patrón Composite (151), ya que una posición de la estructura puede abarcar muchos niveles de agregados anidados. Por tanto, un iterador externo tiene que guardar una ruta a través del Compuesto para saber cuál es el objeto actual. A veces es

más fácil usar un iterador interno. Éste puede guardar la posición actual simplemente llamándose a sí mismo de forma recursiva, por lo que la ruta estará guardada implícitamente en la pila de llamadas.

Si los nodos de un Compuesto tienen una interfaz para ir de un nodo a sus hermanos, padres e hijos, entonces un iterador basado en un cursor puede ofrecer una alternativa mejor. El cursor sólo necesita conocer al nodo actual; puede apoyarse en la interfaz del nodo para recorrer el compuesto.

Los compuestos muchas veces tienen que ser recorridos de más de una manera. Son frecuentes los recorridos en preorden, postorden, enorden y primero-en-anchura. Se puede permitir cada tipo de recorrido con una clase diferente de iterador.

8. *Iteradores nulos*. Un IteradorNulo es un iterador degenerado que ayuda a manejar las condiciones límite. Por definición, un IteradorNulo *siempre* acaba el recorrido; esto es, su operación HaTerminado siempre se evalúa a verdadero.

   IteradorNulo puede facilitar el recorrido de agregados de estructuras de árbol (como los compuestos). En cada punto del recorrido podemos pedirle al elemento actual un iterador para sus hijos. Los elementos agregados devuelven, como norma general, un iterador concreto. Pero los elementos hoja devuelven una instancia de IteradorNulo. Esto nos permite implementar el recorrido sobre la estructura completa de un modo uniforme.

CÓDIGO DE EJEMPLO

Examinaremos la implementación de una clase Lista simple, la cual forma parte de nuestra biblioteca básica (Apéndice C). Se mostrarán dos implementaciones de Iterador, una para recorrer la Lista de principio a fin y otra para recorrerla hacia atrás (la biblioteca sólo permite el primero de estos recorridos). A continuación se muestra

cómo usar estos iteradores y cómo evitar atarse a una determinada implementación. Tras todo eso, se cambia el diseño para asegurar que los iteradores se borran correctamente. El último ejemplo muestra un iterador interno y lo compara con su equivalente externo.

1. *Interfaces Lista e Iterador*. Veamos en primer lugar la parte de la interfaz de Lista que tiene que ver con la implementación de los iteradores. En el Apéndice C puede verse la interfaz completa.

```
template <class Elemento>
class Lista {
public:
     Lista(long       tamaño       =
CAPACIDAD_PREDETERMINADA_LISTA);

     long Contar() const;
     Elemento&  Obtener(long  indice)
const;
     // …
};
```

La clase Lista proporciona, mediante su interfaz pública, un modo razonablemente eficiente de permitir la iteración. Basta con implementar ambos recorridos. De modo que no hay necesidad de dar a los iteradores un acceso restringido a la estructura de datos subyacente; es decir, las clases de los iteradores no son amigas de Lista. Para permitir el uso transparente de diferentes recorridos definimos una clase abstracta Iterador, que define la interfaz del iterador.

```
template <class Elemento>
class Iterador {
public:
     virtual void Primero() = 0;
     virtual void Siguiente() = 0;
     virtual    bool    HaTerminado()
const = 0;
     virtual                 Elemento
ElementoActual() const = 0;
```

```
protected:
     Iterador();
};
```

2. *Implementaciones de las subclases de Iterador*. IteradorLista es una subclase de Iterador.

```
template <class Elemento>
class    IteradorLista    :    public
Iterador<Elemento> {
public:
     IteradorLista(const
Lista<Elemento>* unaLista);
     virtual void Primero();
     virtual void Siguiente();
     virtual    bool    HaTerminado()
const;
     virtual                 Elemento
ElementoActual() const;

private:
     const Lista«Elemento>* _lista;
     long _actual;
};
```

La implementación de IteradorLista es sencilla. Ésta guarda la Lista junto con el índice _actual:

```
template <class Elemento>
IteradorLista«Elemento>::IteradorLista
(
     const Lista<Elemento>* unaLista
) : _lista(unaLista), actual(0) {
}
```

Primero posiciona el iterador en el primer elemento:

```
template «class Elemento»
void
IteradorLista<Elemento>::Primero   ()
```

```
{
     _actual = 0;
}
```

Siguiente avanza hasta el siguiente elemento:

```
template <class Elemento>
void
IteradorLista<Elemento>::Siguiente
() {
     _actual++;
}
```

HaTerminado comprueba si el índice se refiere a un elemento de la Lista:

```
template <class Elemento>
bool
IteradorLista<Elemento>::HaTerminado
() const {
     return   _actual   >=   _lista-
>Contar();
}
```

Por último, ElementoActual devuelve el elemento situado en la posición dada por el índice actual. Sí la iteración ya ha terminado se lanza una excepción IteradorFueraDeLimites:

```
template <class Elemento>
Elemento
IteradorLista<Elemento>::ElementoActual
() const {
     if (HaTerminado()) {
          throw
IteradorFueraDeLimites;
     }
     return                    _lista-
>Obtener(_actual);
```

```
}
```

La implementación de IteradorListaHaciaAtras es idéntica, salvo que su operación Primero posiciona _actual en el final de la lista, y Siguiente va disminuyendo _actual hasta llegar al primer elemento.

3. *Usar los iteradores*. Supongamos que tenemos una Lista de objetos Empleado y queremos imprimir todos los empleados que contiene. La clase Empleado permite esto con una operación Imprimir. Para imprimir la lista, definimos una operación ImprimirEmpleados que toma como parámetros un iterador, al cual usa para recorrer e imprimir la lista.

```
void                ImprimirEmpleados
(Iterador<Empleado*>& i) {
     for        (i.Primero();        !
i.HaTerminado(); i.Siguiente()) {
          i.ElementoActual()-
>Imprimir();
     }
}
```

Puesto que tenemos iteradores para los recorridos hacia atrás y hacia delante, podemos reutilizar esta operación para que muestre los empleados en ambos sentidos.

```
Lista<Empleado*>* empleados;
// …
IteradorLista<Empleado*>
haciaDelante(empleados);
IteradorListaHaciaAtras<Empleado*>
haciaAtras(empleados);

ImprimirEmpleados(haciaDelante);
ImprimirEmpleados(haciaAtras);
```

4. *Evitar ajustarse a una implementación de lista en concreto*.

Pensemos en cómo una variante lista con saltos de Lista afectaría a nuestro código de iteración. Una subclase de Lista, ListaConSaltos, debe proporcionar un IteradorListaConSaltos que implemente la interfaz Iterator. Internamente, el IteradorListaConSaltos tiene que mantener algo más que un índice para hacer la iteración eficientemente. Pero, dado que IteradorListaConSaltos se ajusta a la interfaz de Iterator, la operación ImprimirEmpleados también puede usarse cuando los empleados se guardan en un objeto ListaConSaltos.

```
ListaConSaltos<Empleado*>*
empleados;
// …

IteradorListaConSaltos<Empleado*>
iterador(empleados);
ImprimirEmpleados(iterador);
```

Si bien este enfoque funciona, sería mejor si no tuviéramos que atamos a una determinada implementación de Lista, como por ejemplo ListaConSaltos. Podemos introducir una clase ListaAbstracta para estandarizar la interfaz de lista para diferentes implementaciones de listas, convirtiéndose Lista y ListaConSaltos en subclases de ListaAbstracta.

Para permitir la iteración polimórfica, ListaAbstracta define un método de fabricación CrearIterador, el cual es redefinido por las subclases para devolver su correspondiente iterador.

```
template <class Elemento>
class ListaAbstracta {
public:
     virtual     Iterador<Elemento>*
CrearIterador() const = 0;
     // …
};
```

Una alternativa sería definir una clase mezclable (*mixin*) general, Recorrible, que define la interfaz para crear un iterador. Las clases agregadas pueden combinarse con Recorrible para permitir la iteración polimórfica.

Lista redefine CrearIterador para devolver un objeto IteradorLista:

```
template <class Elemento>
Iterador<Elemento»*
Lista<Elemento>::CrearIterador     ()
const {
     return                          new
IteradorLista<Elemento>(this);
}
```

Ahora nos encontramos en posición de escribir el código para imprimir los empleados independientemente de una representación concreta.

```
//  sólo  sabemos  que  tenemos  una
ListaAbstracta
ListaAbstracta<Empleado*>*
empleados;
// …

Iterador<Empleado*>*    iterador    =
empleados->CrearIterador();
ImprimirEmpleados(*iterador);
delete iterador;
```

5. *Asegurarse de que los iteradores son eliminados*. Nótese que CrearIterador devuelve un objeto iterador recién creado. Somos responsables de borrar dicho objeto. Si nos olvidamos, habremos creado un agujero de memoria. Para facilitar la vida a los clientes, proporcionaremos un PunteroIterador que hace de proxy de un iterador. Se ocupa de limpiar el objeto Iterador cuando éste se sale de ámbito.

PunteroIterador siempre se crea en la pila[53]. C++ se encarga de llamar automáticamente a su constructor cuando se borra el iterador real, punteroIterador sobrecarga tanto operator-> como operator* de modo que un PunteroIterador puede ser tratado exactamente igual que un puntero a un iterador. Los miembros de PunteroIterador se implementan todos en línea; de ese modo no disminuyen el rendimiento.

```
template <class Elemento>
class PunteroIterador {
public:
     PunteroIterador(Iterador<Elemento>*
i): _i (i) { }
     ~PunteroIterador() { delete _i;
}
     Iterador<Elemento>*   operator-
>() { return _i; }
     Iterador<Elemento>& operator*()
{ return *_i; }
private:
     // deshabilita la copia y la
asignación para evitar
     // borrados múltiples de _i:

     PunteroIterador(const
PunteroIterador&);
     PunteroIterador&
operator=(const PunteroIterador&);
private:
     Iterador<Elemento>* _i;
};
```

PunteroIterador nos permite simplificar nuestro código de impresión:

```
ListaAbstracta<Empleado*>*
empleados;
// …

PunteroIterador<Empleado*>
iterador(empleados-
```

```
>CrearIterador());
ImprimirEmpleados(*iterator);
```

6. *Un IteradorLista interno*. Como ejemplo final, echemos un vistazo a una posible implementación de una clase IteradorLista. En este ejemplo es el iterador quien controla la iteración, y quien aplica una iteración a cada elemento.

   El problema en este caso es cómo parametrizar el iterador con la operación que queremos realizar sobre cada elemento. C++ no proporciona funciones anónimas o cierres que otros lenguajes sí proveen para este tipo de tareas. Hay al menos dos opciones: (I) pasar un puntero a una función (global o estática), o (2) apoyarse en la herencia. En el primer caso, el iterador llama, en cada punto de la iteración, a la operación que se le pasó. En el segundo caso, el iterador llama a una operación que una subclase redefine para representar el comportamiento específico.

   Ninguna opción es perfecta. A menudo queremos acumular el estado durante la iteración, y las funciones no están pensadas para eso; tendríamos que usar variables estáticas para recordar el estado. Una subclase de Iterador nos proporciona un lugar apropiado para guardar el estado acumulado, por ejemplo en una variable de instancia. Pero crear una subclase para cada recorrido diferente significa más trabajo.

   Éste es un esbozo de la segunda opción, usando la herencia. Llamaremos al iterador interno un RecorredorLista.

```
template <class Elemento>
Class RecorredorLista {
public:
     RecorredorLista(Lista<Elemento>*
unaLista);
     bool Recorrer();
protected:
```

```
     virtual                          bool
ProcesarElemento(const Elementos.) =
0;
private:
     IteradorLista<Elemento»
_iterador;
};
```

RecorredorLista toma una instancia de Lista como parámetro. Internamente usa un IteradorLista externo para hacer el recorrido. Recorrer comienza el recorrido y llama a ProcesarElemento para cada elemento. El iterador interno puede decidir terminar un recorrido devolviendo false en ProcesarElemento. En caso de que el recorrido haya terminado de forma prematura, Recorrer acaba.

```
template <class Elemento>
RecorredorLista<Elemento>::RecorredorLista
(
     Lista<Elemento>* unaLista
) : _iterador(unaLista) { }

template <class Elemento>
bool
RecorredorLista<Elemento>::Recorrer
() {
     bool resultado = false;

     for (
          _iterador.Primero();
          !_iterador.HaTerminado();
          _iterador.Siguiente()
     ) {
          resultado                   =
ProcesarElemento(_iterador.ElementoActual());

          if (resultado == false) {
               break;
          }
     }
     return resultado;
}
```

Usemos ahora un RecorredorLista para imprimir los primeros 10 empleados de nuestra lista de empleados. Para hacer esto tenemos que heredar de RecorredorLista y redefinir ProcesarElemento. Contamos el numero de empleados mostrados en una variable de instancia _contador.

```
class ImprimirNEmpleados : public
RecorredorLista<Empleado*> {
public:
     ImprimirNEmpleados(Lista<Empleado*>*
unaLista, int n) :
          RecorredorLista<Empleado*>(unaLista),
          _total(n), _contador(0) {
}

protected:
     bool ProcesarElemento(Empleado*
const&);
private:
     int _total;
     int _contador;
};

bool
ImprimirNEmpleados::ProcesarElemento
(Empleado* const& e) {
     _contador++;
     e->Imprimir();
     return _contador < _total;
}
```

Así es como ImprimirNEmpleados imprime los primeros 10 empleados de la lista:

```
Lista<Empleado*»* empleados;
// …
ImprimirNEmpleados     pa(empleados,
10);
pa.Recorrer();
```

Nótese cómo el cliente no especifica el bucle de iteración. Toda la lógica de iteración puede ser reutilizada. Ésta es la principal ventaja de un operador interno. Es algo más de trabajo que con un iterador externo, ya que tenemos que definir una nueva clase. Compárese esto con el uso de un iterador externo:

```
IteradorLista<Empleado*>
i(empleados);
int contador = 0;

for  (i.Primero();  !i.HaTerminado();
i.Siguiente()) {
     contador++;
     i.ElementoActual()->Imprimir();

     if (contador >= 10) {
          break;
     }
}
```

Los iteradores internos pueden encapsular diferentes tipos de iteración. Por ejemplo, RecorredorListaConFiltro encapsula una iteración que procesa sólo aquellos elementos que verifican una comprobación:

```
template <class Elemento>
class RecorredorListaConFiltro {
public:
     RecorredorListaConFiltro(Lista<Elemento>*
unaLista);
     bool Recorrer();
protected:
     virtual                         bool
ProcesarElemento(const  Elemento&)  =
0;
     virtual                         bool
ComprobarElemento(const  Elemento&)  =
0;
private:
     IteradorLista<Elemento>
_iterador;
```

```
};
```

Esta interfaz es la misma que la de RecorredorLista salvo por una función miembro añadida, ComprobarElemento, que define la comprobación. Las subclases redefinen ComprobarElemento para especificar la comprobación.

Recorrer decide continuar el recorrido basándose en el resultado de la comprobación:

```
template <class Elemento>
void
RecorredorListaConFiltro<Elemento>::Recorrer
() {
     bool resultado = false;

     for (
          _Iterador.Primero();
          !_iterador.HaTerminado();
          _iterador.Siguiente()
     ) {
          if
(ComprobarElemento(_iterador.ElementoActual()))
{
               resultado           =
ProcesarElemento(_iterador.ElementoActual());
               if    (resultado    ==
false) {
                    break;
               }
          }
     }
     return resultado;
}
```

Una variante de esta clase podría definir Recorrer para que terminase si al menos un elemento satisface la comprobación **[54]** .

USOS CONOCIDOS

Los iteradores son frecuentes en los sistemas orientados a objetos.

La mayoría de las bibliotecas de clases ofrecen iteradores de una forma u otra.

El siguiente es un ejemplo de los componentes de Booch [Boo94], una popular biblioteca de clases de colecciones. Proporciona dos implementaciones de una cola, una para un tamaño fijo y otra que puede crecer dinámicamente. La interfaz de la cola la define una clase abstracta Queue. Para permitir iterar polimórficamente sobre las diferentes implementaciones de colas, el iterador de la cola se implementa en términos de la interfaz de la clase abstracta Queue. Esta variación tiene la ventaja de que no se necesita un método de fabricación para pedirle a las implementaciones de colas su iterador apropiado. No obstante, en necesario que la interfaz de la clase abstracta Queue sea los bastante potente como para implementar de manera eficiente el iterador.

En Smalltalk no hay que definir explícitamente los iteradores. Las clases de colecciones estándar Bag, Set, Dictionary, OrderedCollection, String, etc.) definen un método do: como un iterador interno, el cual toma un bloque (es decir, un cierre) como parámetro. Cada parámetro de la colección se liga a la variable local del bloque; a continuación se ejecuta el bloque. Smalltalk también incluye un conjunto de clases Stream que permiten una interfaz similar a un iterador. ReadStream es básicamente un iterador, y puede actuar como un iterador externo para todas las colecciones secuenciales. No hay iteradores externos estándar para las colecciones no secuenciales como Set y Dictionary.

Las clases contenedoras de ETF+ proporcionan los iteradores polimórficos y el Proxy de limpieza descritos anteriormente [WGM88]. Las clases del framework de editores gráficos usan iteradores basados en cursores [VL90],

ObjectWindows 2.0 [Bor94] provee una jerarquía de clases de iteradores para contenedores. Se puede iterar sobre diferentes tipos de contenedores del mismo modo. La sintaxis de iteración de ObjectWindow se basa en sobrecargar el operador de incremento postfijo, + +, para avanzar en la iteración.

PATRONES RELACIONADOS

Composite (151): los iteradores suelen aplicarse a estructuras

recursivas como los compuestos.

Factory Method (99): los iteradores polimórficos se basan en métodos de fabricación para crear instancias de las subclases apropiadas de Iterator.

El patrón Memento (261) suele usarse conjuntamente con el patrón Iterator. Un iterador puede usar un memento para representar el estado de una iteración. El iterador almacena el memento internamente.

MEDIATOR (Mediador)

PROPÓSITO

Define un objeto que encapsula cómo interactúan una serie de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.

MOTIVACIÓN

Los diseñadores orientados a objetos promueven la distribución de comportamiento entre objetos. Dicha distribución puede dar lugar a una estructura de objetos con muchas conexiones entre ellos; en el peor de los casos, cada objeto acaba por conocer a todos los demás.

Aunque dividir un sistema en muchos objetos suele mejorar la reutilización, la proliferación de interconexiones tiende a reducir ésta de nuevo. Tener muchas interconexiones hace que sea menos probable que un objeto pueda funcionar sin la ayuda de otros —el sistema se comporta como si fuera monolítico—. Más aún, puede ser difícil cambiar el comportamiento del sistema de manera significativa, ya que él comportamiento se encuentra distribuido en muchos objetos. Como resultado, podemos vernos forzados a definir muchas subclases para personalizar el comportamiento del sistema.

Sea, por ejemplo, la implementación de cuadros de diálogo en una interfaz gráfica de usuario. Un cuadro de diálogo usa una ventana para presentar una colección de útiles[55] tales como botones, menús y campos de entrada, como se muestra a continuación;

Fuente

The quick brown fox...

Familia new century schoolbook

avant garde
chicago
courier
helvetica
new century schoolbook
palatino
times roman
zapf dingbats

Peso normal negrita seminegrita

Estilo normal cursiva inclinada

Tamaño 34 pt condensada

Cancelar Aceptar

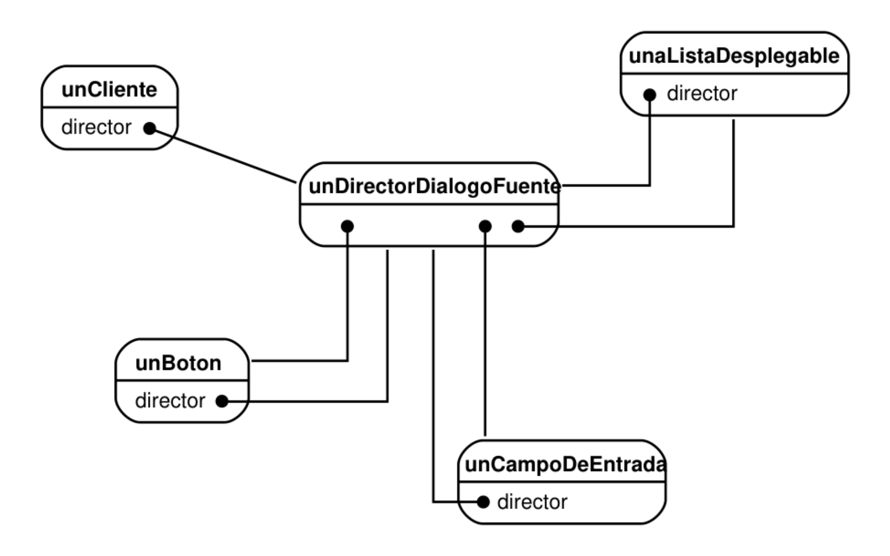
Muchas veces hay dependencias entre los útiles del diálogo. Por ejemplo, un botón está desactivado cuando está vacío un campo de entrada determinado. Seleccionar una entrada de una lista de opciones, llamada **lista desplegable** (*listbox*), puede cambiar el contenido de un campo de entrada. A la inversa, teclear texto en el campo de entrada puede seleccionar automáticamente una o más entradas de la lista desplegable. En cuanto haya texto en el campo de entrada pueden activarse otros botones para permitir al usuario hacer algo con dicho texto, como cambiar o borrar aquello a lo que se refiere.

Distintos cuadros de diálogo tendrán distintas dependencias entre útiles. Así, aunque los diálogos muestran los mismos tipos de útiles, no pueden reutilizar directamente las clases de útiles de las que se dispone; se ven obligados a personalizarlas para que reflejen las dependencias específicas de cada diálogo. Personalizarlas una a

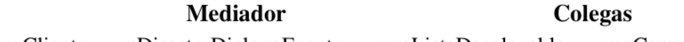
una mediante la herencia sería tedioso, ya que hay muchas clases involucradas.

Estos problemas pueden ser evitados encapsulando el comportamiento colectivo en un objeto aparte llamado **mediador**. Un mediador es responsable de controlar y coordinar las interacciones entre un grupo de objetos. El mediador hace las veces de un intermediario que evita que los objetos del grupo se refieran unos a otros explícitamente. Los objetos sólo conocen al mediador, reduciendo así el número de interconexiones.

Por ejemplo, **DirectorDialogoFuente** podría ser el mediador entre los útiles de un cuadro de diálogo. Un objeto DirectorDialogoFuente conoce a los útiles de un diálogo y coordina su interacción. Funciona como un concentrador **[56]** de comunicaciones para útiles:



El siguiente diagrama de interacción ilustra cómo cooperan los objetos para manejar un cambio en la selección de una lista:

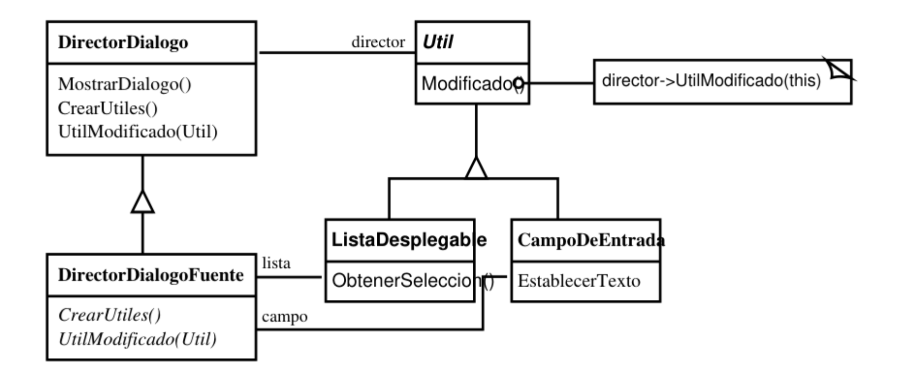


Ésta es la sucesión de eventos que tienen lugar cuando la selección de una lista pasa a un campo de entrada:

1. La lista desplegable le dice a su director que ha cambiado.
2. El director obtiene la selección de la lista.
3. El director le pasa la selección al campo de entrada.
4. Ahora que el campo de entrada contiene algo de texto, el director activa uno o varios botones que permitan inicializar una acción (por ejemplo, “negrita” o “cursiva”).

Nótese cómo el director media entre la lista y el campo de entrada. Los útiles se comunican unos con otros sólo indirectamente, a través del director. No tienen que saber nada de los otros; al único al que conocen es al director. Además, dado que el comportamiento se encuentra localizado en una clase, puede cambiarse o reemplazarse extendiendo o sustituyendo esa clase.

Así es como la abstracción DirectorDialogoFuente puede integrarse en una biblioteca de clases:



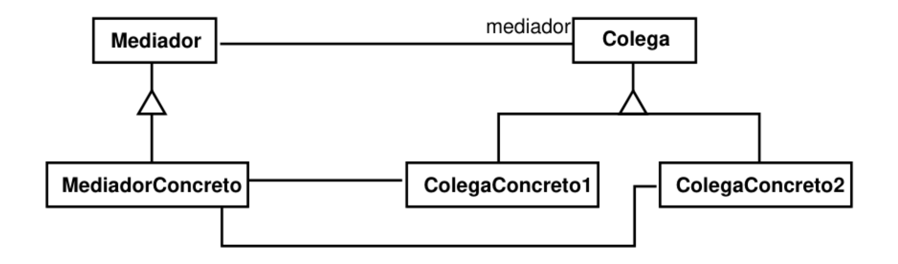
DirectorDialogo es una clase abstracta que define el comportamiento general de un diálogo. Los clientes llaman a la operación MostrarDialogo para mostrar el diálogo en la pantalla. CrearUtiles es una operación abstracta que crea los útiles de un diálogo. UtilModificado es otra operación abstracta a la que llaman los útiles para informar a su director de que han cambiado. Las subclases de DirectorDialogo redefinen CrearUtiles para crear los útiles apropiados, así como UtilModificado para manejar los cambios.

APLICABILIDAD

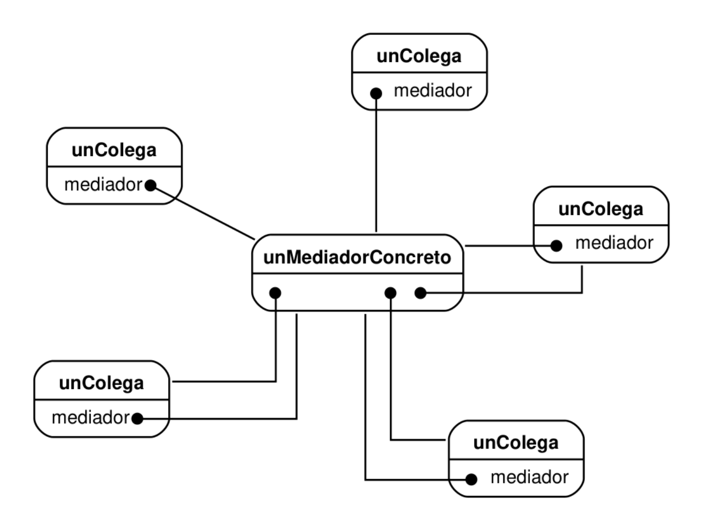
Úsese el patrón Mediator cuando

- un conjunto de objetos se comunican de forma bien definida, pero compleja. Las interdependencias resultantes no están estructuradas y son difíciles de comprender.
- es difícil reutilizar un objeto, ya que éste se refiere a otros muchos objetos, con los que se comunica.
- un comportamiento que está distribuido entre varias clases debería poder ser adaptado sin necesidad de una gran cantidad de subclases.

ESTRUCTURA



Una estructura de objetos típica podría parecerse a ésta:



PARTICIPANTES

- **Mediador** (DirectorDialogo)
  - define una interfaz para comunicarse con sus objetos Colega.
- **MediadorConcreto** (DirectorDialogoFuente)
  - implemento el comportamiento cooperativo coordinando objetos Colega.
  - conoce a sus Colegas.
- **clases Colega** (ListaDesplegable, CampoDeEntrada)
  - cada clase Colega conoce a su objeto Mediador.
  - cada Colega se comunica con su mediador cada vez que, de no existir éste, se hubiera comunicado con otro

Colega.

COLABORACIONES

Los Colegas envían y reciben peticiones a través de un Mediador. El mediador implementa el comportamiento cooperativo encaminando estas peticiones a los Colegas apropiados.

CONSECUENCIAS

El patrón Mediator tiene las siguientes ventajas e inconvenientes:

1. *Reduce la herencia*. Un mediador localiza el comportamiento que de otra manera estaría distribuido en varios objetos. Para cambiar este comportamiento sólo es necesario crear una subclase del Mediador; las clases Colega pueden ser reutilizadas tal cual.
2. *Desacopla a los Colegas*. Un mediador promueve un bajo acoplamiento entre Colegas. Las clases Colega pueden usarse y modificarse de forma independiente.
3. *Simplifica los protocolos de los objetos*. Un mediador sustituye interacciones muchos-a-muchos por interacciones uno-a-muchos entre el mediador y sus Colegas. Las relaciones uno-a-muchos son más fáciles de comprender, mantener y extender.
4. *Abstrae cómo cooperan los objetos*. Hacer de la mediación un concepto independiente y encapsularla en un objeto permite centrarse en cómo interactúan los objetos en vez de en su comportamiento individual. Eso ayuda a clarificar cómo interactúan los objetos de un sistema.
5. *Centraliza el control*. El patrón Mediator cambia complejidad de interacción por complejidad en el mediador. Dado que un mediador encapsula protocolos, puede hacerse más complejo que cualquier Colega individual. Esto puede hacer del mediador un monolito difícil de mantener.

IMPLEMENTACIÓN

Son pertinentes los siguientes detalles de implementación para el patrón Mediator:

1. *Omitir la clase abstracta Mediador*. No es necesario definir una clase abstracta Mediador cuando los Colegas sólo trabajan con un mediador. El acoplamiento abstracto proporcionado por la clase Mediador permite que los Colegas trabajen con diferentes subclases de Mediador, y viceversa.
2. *Comunicación Colega-Mediador*. Los Colegas tienen que comunicarse con su mediador cuando tiene lugar un evento de interés. Un enfoque es implementar el Mediador como un Observador, usando el patrón Observer (269). Las clases Colega hacen de Sujeto, enviando notificaciones al mediador cada vez que cambia su estado. El mediador responde propagando los efectos del cambio a otros Colegas.

Otro enfoque define en el Mediador una interfaz de notificación especializada que permite a los Colegas ser más directos en su comunicación. Smalltalk/V para Windows usa una forma de delegación: cuando se comunica con el mediador, un Colega se pasa a sí mismo como parámetro, permitiendo al mediador identificar al emisor. El Código de Ejemplo usa este enfoque y la implementación de Smalltalk/V se examina más adelante en los Usos Conocidos.

CÓDIGO DE EJEMPLO

Usaremos un DirectorDialogo para implementar el cuadro de diálogo de selección del tipo fuente que se mostró en la Motivación. La clase abstracta DirectorDialogo define la interfaz para los directores.

```
class DirectorDialogo {
public:
     virtual ~DirectorDialogo();

     virtual void MostrarDialogo();
     virtual void UtilModificado(Util*) = 0;

protected:
     DirectorDialogo();
     virtual void CrearUtiles() = 0;
```

```
};
```

Util es la clase base abstracta para los útiles. Un útil conoce a su director.

```
class Util {
public:
     Util(DirectorDialogo*);
     virtual void Modificado();

     virtual  void  ManejarRaton(EventoRaton&
evento);
     // …
private:
     DirectorDialogo* _director;
};
```

Modificado llama a la operación UtilModificado del director. Los útiles llaman a UtilModificado sobre su director para informarle de un evento significativo.

```
void Util::Modificado () {
     _director->UtilModificado(this);
}
```

Las subclases de DirectorDialogo redefinen UtilModificado para afectar a los útiles apropiados, El útil pasa una referencia a sí mismo como parámetro a UtilModificado, para que el director pueda identificar al útil modificado. Las subclases de DirectorDialogo redefinen el método virtual puro CrearUtiles para construir los útiles del diálogo.

ListaDesplegable, CampoDeEntrada y Boton son subclases de Util para elementos especializados de la interfaz de usuario. ListaDesplegable proporciona una operación ObtenerColeccion para obtener la selección actual, y la operación EstablecerTexto de CampoDeEntrada permite poner texto en el campo.

```
class ListaDesplegable : public Util {
public:
     ListaDesplegable(DirectorDialogo*);

     virtual const char* ObtenerSeleccion();
     virtual                               void
EstablecerLista(Lista<char*>* elementos);
     virtual  void  ManejarRaton(EventoRaton&
evento);
     // …
};

class CampoDeEntrada : public Util {
public:
     CampoDeEntrada(DirectorDialogo*);

     virtual void EstablecerTexto(const char*
texto);
     virtual const char* ObtenerTexto();
     virtual  void  ManejarRaton(EventoRaton&
evento);
     // …
};
```

Boton es un útil sencillo que llama a Modificado cada vez que es pulsado. Esto se buce en la implementación de ManejarRaton:

```
class Boton : public Util {
public:
     Boton(DirectorDialogo*);

     virtual void EstablecerTexto(const char*
texto);
     virtual  void  ManejarRaton(EventoRaton&
evento);
     // …
};

void     Boton::ManejarRaton     (EventoRaton&
evento)  (
     // …
     Modificado();
}
```

La clase DirectorDialogoFuente media entre los útiles del cuadro de

diálogo. DirectorDialogoFuente es una subclase de DirectorDialogo:

```
class    DirectorDialogoFuente    :    public
DirectorDialogo {
public:
     DirectorDialogoFuente();
     virtual ~DirectorDialogoFuente();
     virtual void UtilModificado(Util*);

     protected:
     virtual void CrearUtiles();
private:
     Boton* _aceptar;
     Boton* _cancelar;
     ListaDesplegable* _fuenteLista;
     CampoDeEntrada* _nombreFuente;
};
```

DirectorDialogoFuente se ocupa de los útiles que muestra. Redefine CrearUtiles para crear los útiles e inicializar sus referencias para que apunten a él:

```
void DirectorDialogoFuente::CrearUtiles () {
     _aceptar = new Boton(this);
     _cancelar = new Boton(this);
     _fuenteLista             =            new
ListaDesplegable(this);
     _nombreFuente             =           new
CampoDeEntrada(this);
     // rellenar la lista con los nombres de
fuentes disponibles
     // ensambla los útiles en el diálogo
}
```

UtilModificado garantiza que los útiles trabajen juntos adecuadamente:

```
void DirectorDialogoFuente::UtilModificado (
     Util* elUtilModificado
) {
     if (elUtilModificado == _fuenteLista) (
```

```
            _nombreFuente->EstablecerTexto(
                 _fuenteLista-
>ObtenerSeleccion());

            }  else  if  (elUtilModificado  ==
_aceptar) {
                 // aplicar el cambio de fuente
y cerrar el diálogo
                 // …

            }  else  if  (elUtilModificado  ==
_cancelar) (
                 // cerrar el diálogo
            }
       }
```

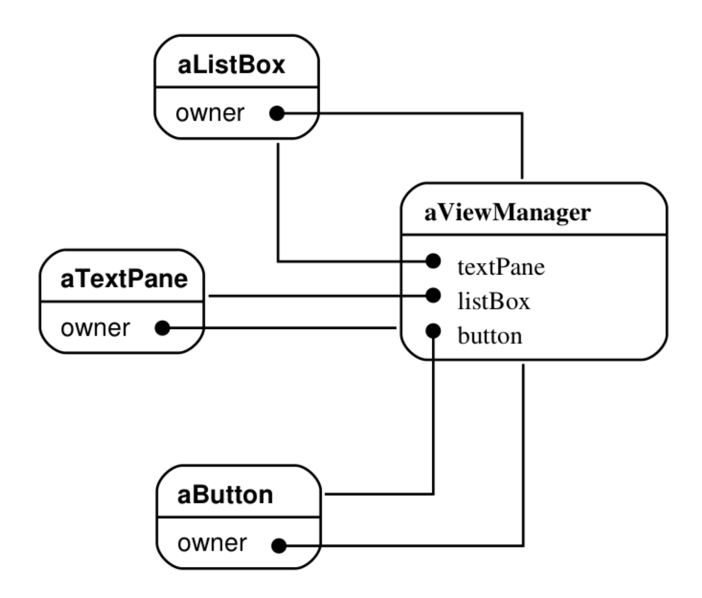
La complejidad de UtilModificado crece proporcionalmente a la complejidad del diálogo. Los diálogos grandes son indeseables por otras razones, por supuesto, pero la complejidad del mediador podría mitigar los beneficios del patrón en otras aplicaciones.

USOS CONOCIDOS

Tanto ET++ [WGM88] como la biblioteca de clases THINK C [Sym93b] usan objetos parecidos a directores como mediadores entre los útiles de los diálogos.

La arquitectura de Smalltalk/V para Windows está basada en una estructura de mediadores [LaL94]. En ese entorno, una aplicación consiste en un objeto Window que contiene un conjunto de paneles. La biblioteca contiene varios objetos Pane predefinidos; ejemplos de éstos son TextPane, ListBox, Button, etcétera. Estos paneles pueden usarse sin necesidad de heredar de ellos. Un desarrollador de aplicaciones sólo hereda de ViewManager, una clase que es la responsable de realizar la coordinación entre paneles. ViewManager es el Mediador, y cada panel sólo conoce a su gestor de vistas, considerado como el “propietario” del panel. Los paneles no se refieren unos a otros directamente.

El siguiente diagrama de objetos muestra una instantánea de una aplicación en tiempo de ejecución:



Smalltalk/V usa un mecanismo de eventos para la comunicación PaneViewManager. Un panel genera un evento cuando quiere obtener información del mediador o cuando quiere informar a éste de que ha ocurrido algo significativo. Un evento define un símbolo (por ejemplo, #select) que lo identifica. Para manejar el evento, el gestor de vistas registra un selector de método con el panel. Este selector es el manejador del evento; será invocado cada vez que ocurra un evento.

El código siguiente muestra un esbozo de cómo se crea un objeto ListPane dentro de una subclase de ViewManager y cómo ViewManager registra un manejador de eventos para el evento «select:

```
self addSubpane: (ListPane new
          paneName: 'miListPane';
          owner: self;
          when:       #select       perform:
#listSelect:).
```

Otra aplicación del patrón Mediator es coordinar actualizaciones complejas. Un ejemplo de esto es la clase GestorDeCambios mencionada en el patrón Observer (269). GestorDeCambios hace de

mediador entre sujetos y observadores para evitar actualizaciones redundantes. Cuando un objeto cambia, se notifica al GestorDeCambios, quien coordina la actualización notificando a su vez a los objetos dependientes.

Una aplicación similar aparece en el framework de dibujo Unidraw [VL90], que usa una clase llamada CSolver para hacer cumplir las restricciones de conectividad entre “conectores”. Los objetos de los editores gráficos pueden unirse entre sí de diferentes formas. Los conectores son útiles en aplicaciones que mantienen la conectividad automáticamente, como editores de diagramas y sistemas de diseño de circuitos. CSolver es un mediador entre conectores, que resuelve las restricciones de conectividad y actualiza las posiciones de los conectores para que las reflejen adecuadamente.

PATRONES RELACIONADOS

El patrón Facade (171) difiere del Mediator en que abstrae un subsistema de objetos para proporcionar una interfaz más conveniente. Su protocolo es unidireccional; es decir, los objetos Fachada hacen peticiones a las clases del subsistema pero no a la inversa. Por el contrario, el patrón Mediator permite un comportamiento cooperativo que no es proporcionado por los objetos Colegas y el protocolo es multidireccional.

Los Colegas pueden comunicarse con el mediador usando el patrón Observer (269).

MEMENTO (Recuerdo)

PROPÓSITO

Representa y externaliza el estado interno de un objeto sin violar, la encapsulación, de forma que éste puede volver a dicho estado más tarde.

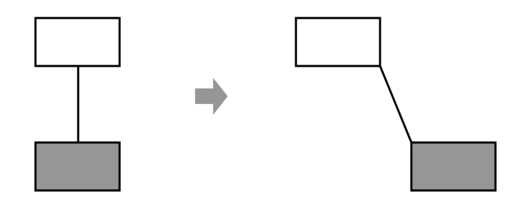
TAMBIÉN CONOCIDO COMO

Token [57]

MOTIVACIÓN

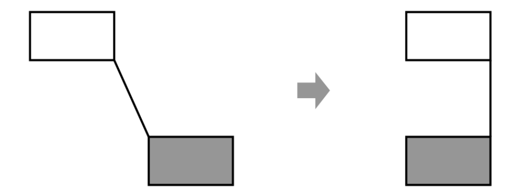
A veces es necesario guardar el estado interno de un objeto. Esto es necesario cuando se implementan casillas de verificación o mecanismos de deshacer que permiten a los usuarios anular operaciones provisionales y recuperarse de los errores. Debe guardarse información del estado en algún sitio para que los objetos puedan volver a su estado anterior. Pero los objetos normalmente encapsulan parte de su estado, o todo, haciéndolo inaccesible a otros objetos e imposible de guardar externamente. Exponer este estado violaría la encapsulación, lo que puede comprometer la fiabilidad y extensibilidad de la aplicación.

Pensemos, por ejemplo, en un editor gráfico que permite conectar objetos. Un usuario puede conectar dos rectángulos con una línea, y los rectángulos permanecen conectados cuando el usuario mueve cualquiera de ellos. El editor garantiza que la línea se estira para mantener la conexión.



Una forma habitual de mantener relaciones de conectividad entre objetos es mediante un sistema de resolución de problemas. Podemos encapsular esta funcionalidad en un objeto ResolventeDeRestricciones. ResolventeDeRestricciones almacena las conexiones a medida que se van creando éstas y genera ecuaciones matemáticas que las describen. Cada vez que el usuario hace una conexión o modifica el diagrama resuelve dichas ecuaciones. ResolventeDeRestricciones usa los resultados de sus cálculos para volver a colocar los gráficos de forma que mantengan las conexiones adecuadas.

Permitir que se puedan deshacer ciertas operaciones en esta aplicación no es tan sencillo como en un principio podría parecer. Una manera obvia de deshacer una operación de movimiento es guardar la distancia que se ha, movido el objeto desde su posición original y mover éste hacia atrás una distancia equivalente. Sin embargo, esto no garantiza que todos los objetos aparezcan como estaban antes. Supongamos que hay algo de holgura en la conexión. En ese caso, mover el rectángulo hacia atrás, a su posición original, no producirá necesariamente el efecto deseado.



En general, la interfaz pública de ResolventeDeRestricciones podría no ser suficiente para permitir revertir con precisión sus efectos sobre otros objetos. El mecanismo de deshacer debe trabajar más estrechamente con ResolventeDeRestricciones para restablecer el estado previo, pero, por otro lado, también deberíamos evitar exponer al mecanismo de deshacer las interioridades de ResolventeDeRestricciones.

Este problema se puede solucionar con el patrón Memento. Un memento es un objeto que almacena una instantánea del estado interno de otro objeto —el creador del memento—. El mecanismo de deshacer solicitará un memento al creador cuando necesite comprobar el estado de éste. El creador inicializa el memento con información que representa su estado actual. Sólo el creador puede almacenar y recuperar información del memento —el memento es “opaco” a otros objetos—.

En el ejemplo del editor gráfico que se acaba de describir, el ResolventeDeRestricciones puede actuar como un creador. La siguiente secuencia de eventos representa el proceso de deshacer:

1. El editor solicita un memento al ResolventeDeRestricciones como un efecto lateral de la operación de mover.
2. El ResolventeDeRestriccioncs crea y devuelve un memento, en este caso una instancia de una clase EstadoDelResolvente. Un memento del EstadoDelResolvente contiene estructuras de datos que describen el estado actual de las ecuaciones y variables internas del ResolventeDeRestricciones.
3. Más tarde, cuando el usuario deshace la operación de mover, el editor le devuelve al ResolventeDeRestricciones el EstadoDelResolvente.
4. Dependiendo de la información del EstadoDelResolvente, el ResolventeDeRestricciones cambia sus estructuras de datos internas para devolver sus ecuaciones y variables exactamente a su estado anterior.

Este acuerdo permite al ResolventeDeRestriccioncs confiar a otros objetos la información que necesita para volver a un estado previo sin exponer sus estructuras y representaciones internas.

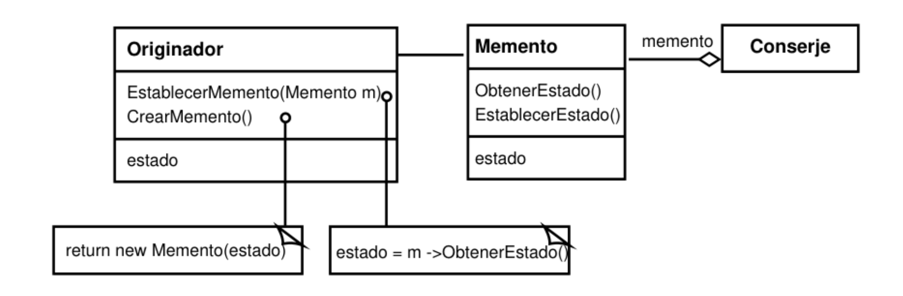
APLICABILIDAD

Úsese el patrón Memento cuando

- hay que guardar una instantánea del estado de un objeto (o de parte de éste) para que pueda volver posteriormente a ese estado, y
- una interfaz directa para obtener el estado exponga detalles

de implementación y rompa la encapsulación del objeto

ESTRUCTURA

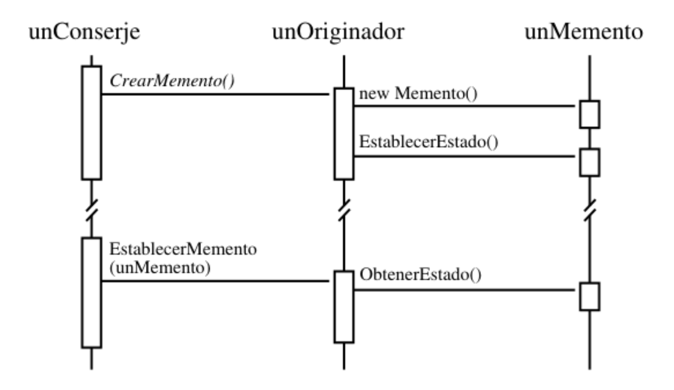


PARTICIPANTES

- **Memento** (EstadoDelResolvente)
    - guarda el estado interno del objeto Creador. El memento puede guardar tanta información del estado interno del creador como sea necesario a discreción del creador
    - protege frente a accesos de otros objetos que no sean el creador. Los mementos tienen realmente dos interfaces. El Conserje ve una interfaz *reducida* del Memento —sólo puede pasar el memento a otros objetos—. El Creador, por el contrario, ve una interfaz *amplia*, que le permite acceder a todos los datos necesarios para volver a su estado anterior. Idealmente, sólo el creador que produjo el memento estaña autorizado a acceder al estado interno de éste.
- **Creador** (ResolventeDeRestricciones)
    - crea un memento que contiene una instantánea de su estado interno actual.
    - usa el memento para volver a su estado anterior.
- **Conserje** (mecanismo de deshacer)
    - es responsable de guardar en lugar seguro el memento.
    - nunca examina los contenidos del memento, ni opera sobre ellos.

COLABORACIONES

- Un conserje solicita un memento a un creador, lo almacena durante un tiempo y se lo devuelve a su creador, tal y como ilustra el siguiente diagrama de interacción:



A veces el conserje no devolverá el memento a su creador, ya que el creador podría no necesitar nunca volver a un estado anterior.

- Los mementos son pasivos. Sólo el creador que creó el memento asignará o recuperará su estado.

CONSECUENCIAS

El patrón Memento tiene varias consecuencias:

1. *Preservación de los límites de la encapsulación*. El memento evita exponer información que sólo debería ser gestionada por un creador, pero que sin embargo debe ser guardada fuera del creador. El patrón oculta a otros objetos las interioridades, potencialmente complejas, del Creador, preservando así los límites de la encapsulación.
2. *Simplifica al Creador*. En otros diseños que persiguen conservar la encapsulación, el Creador mantiene las versiones de su estado interno que han sido solicitadas por los clientes. Eso asigna toda la responsabilidad de gestión del

almacenamiento al Creador. Que sean los clientes quienes gestionen el estado que solicitan simplifica al Creador y evita que los clientes tengan que notificar a los creadores cuando han acabado.

3. *El uso de mementos puede ser costoso*. Los mementos podrían producir un coste considerable si el Creador debe copiar grandes cantidades de información para guardarlas en el memento o si los clientes crean y devuelven mementos a su creador con mucha frecuencia. A menos que encapsular y restablecer el estado del Creador sea poco costoso, el patrón podría no ser apropiado. Véase la discusión acerca de los cambios increméntales en la sección de Implementación.
4. *Definición dé interfaces reducidas y amplias*. En algunos lenguajes puede ser difícil garantizar que sólo el creador acceda al estado del memento.
5. *Costes ocultos en el cuidado de los mementos*. Un conserje es responsable de borrar los mementos que custodia. Sin embargo, el conserje no sabe cuánto estado hay en un memento. De ahí que un conserje que debería ser ligero pueda provocar grandes costes de almacenamiento cuando debe guardar mementos.

IMPLEMENTACIÓN

Éstas son dos cuestiones a considerar a la hora de implementar el patrón Memento:

1. *Soporte del lenguaje*. Los mementos tienen dos interfaces: una amplia para los creadores y otra reducida para otros objetos. Lo ideal sería que el lenguaje de implementación permitiese dos niveles de protección estática. C++ permite hacer esto haciendo que Creador sea una clase amiga del Memento y haciendo privada a la interfaz amplia del Memento. Sólo la interfaz reducida debería ser declarada pública. Por ejemplo:

```
class Estado;
```

```
class Creador {
public:
     Memento* CrearMemento();
     void    EstablecerMemento(const
Memento*);
     // …
private:
     Estado* _estado;
     //    estructuras    de    datos
internas
     // …
};

class Memento {
public:
     // interfaz pública reducida
     virtual ~Memento();
private:
     // miembros privados accesibles
sólo pare el Creador
     friend class Creador;
     Memento();

     void EstablecerEstado(Estado*);
     Estado* ObtenerEstado();
     // …
private:
     Estado* _estado;
     // …
};
```

2. *Guardar sólo los cambios incrementales*. Cuando los mementos se crean y se devuelven a su creador en una secuencia predecible, el Memento puede guardar únicamente el *cambio* con respecto al estado interno del creador.

   Por ejemplo, las órdenes de deshacer de un historial pueden usar mementos para asegurar que las órdenes vuelven a su estado original exacto cuando se deshacen (*véase* el patrón Command (215)). El historial define un orden concreto en el que las órdenes pueden deshacerse y repetirse. Eso significa que los mementos pueden guardar únicamente el cambio producido por una orden, en vez de)

estado completo de cada objeto al que afecta. En el ejemplo de la sección de Motivación, el resolvente de problemas podría guardar sólo aquellas estructuras internas que cambian para que las líneas sigan uniendo a los rectángulos, en vez de guardar las posiciones absolutas de estos objetos.

CÓDIGO DE EJEMPLO

El código C++ que aquí se muestra ilustra el ejemplo del ResolventeDeRestriccioncs que se discutió anteriormente. Hemos usado objetos OrdenMover (*véase* Command (215)) para (des)hacer la traslación de un objeto gráfico de una posición a otra. El editor gráfico llama a la operación Ejecutar de la orden para mover un objeto gráfico, y a Deshacer para deshacer el movimiento. La orden guarda su destino, la distancia movida y una instancia de MementoDelResolventeDeProblemas, un memento que contiene estado del resolvente de problemas.

```
class Grafico;
     // clase base para los objetos gráficos
del editor gráfico
class OrdenMover {
public:
     OrdenMover(Grafico*    destino,    const
Punto& incremento);
     void Ejecutar();
     void Deshacer();
private:
     MementoDelResolventeDeRestricciones*
_estado;
     Punto _incremento;
     Gráfico* _destino;
};
```

Las restricciones de Conexión son establecidas por la clase ResolventeDeRestricciones. Su principal función miembro es Resolver, que resuelve las restricciones registradas con la operación AnadirRestriccion. Para permitir deshacer, el estado del ResolventeDeRestricciones puede externalizarse con CrearMemento

en una instancia de MementoDelResolventeDeRestricciones. El resolvente de restricciones puede volver a un estado previo llamando a EstablecerMemento. ResolventeDeRestricciones es un Singleton (119).

```
class ResolventeDeRestricciones {
public:
     static        ResolventeDeRestricciones*
Instancia();

     void Resolver();
     void AnadirRestriccion(
          Grafico*        principioConexion,
Grafico* finConexion
     );
     void EliminarRestriccion(
          Grafico*        principioConexion,
Grafico* finConexion
     );

     MementoDelResolventeDeRestricciones*
CrearMemento();
     void
EstablecerMemento(MementoDelResolventeDeRestricciones*);
private:
     // estado no trivial y operaciones para
hacer cumplir
     //la semántica de las conexiones
};

class MementoDelResolventeDeRestricciones {
public:
     virtual
~MementoDelResolventeDeRestricciones();
private:
     friend class ResolventeDeRestricciones;
     MementoDelResolventeDeRestricciones();
     //  estado  privado  del  resolvente  de
restricciones
};
```

Dadas estas interfaces, podemos implementar los miembros Ejecutar y Deshacer de OrdenMover como sigue:

```
void OrdenMover::Ejecutar () {
     ResolventeDeRestricciones* resolvente =
ResolventeDeRestricciones::Instancia();
     _estado = resolvente->CrearMemento(); //
crea un memento
     _destino->Mover(_incremento);
     resolvente->Resolver();
}

void OrdenMover::Deshacer () {
     ResolventeDeRestricciones* resolvente =
     ResolventeDeRestricciones::Instancia();
     _destino->Mover(-_incremento);
     resolvente->EstablecerMemento(_estado);
// vuelve al estado anterior
     resolvente->Resolver();
}
```

Ejecutar adquiere un memento del ElementoDelResolventeDeRestricciones antes de mover el gráfico. Deshacer mueve el gráfico hacia atrás, devuelve el resolvente de restricciones a su estado anterior y, por último, le dice al resolvente de restricciones que resuelva sus restricciones.

USOS CONOCIDOS

El ejemplo anterior está tomado del soporte para conectividad de Unidraw a través de su clase CSolver [VL90].

Las colecciones de Dylan [App92] proporcionan una interfaz de iteración que refleja el patrón Memento. Estas colecciones tienen la noción de un objeto "estado", el cual es un memento que representa el estado de la iteración. Cada colección puede representar el estado actual de la iteración en la forma que prefiera; dicha representación permanece completamente oculta a los clientes. El enfoque de Dylan para la iteración podría trasladarse así a C++:

```
template <class Elemento>
class Coleccion {
public:
     coleccion();

     EstadoIteracion* CrearEstadoInicial();
```

```
     void Siguiente(EstadoIteracion*);
     bool HaTerminado(const EstadoIteracion*)
const;
     Elemento           ElementoActual(const
EstadoIteracion*) const;
     EstadoIteracion*           Copiar(const
EstadoIteracion*) const;

     void Insertar(const Elemento&);
     void Eliminar(const Elemento&);
     // …
};
```

CrearEstadoInicial devuelve un objeto EstadoIteracion inicializado para la colección. Siguiente hace avanzar el objeto estado a la siguiente posición en la iteración; en realidad lo que hace es incrementar el índice de la iteración. HaTerminado devuelve true si Siguiente ha avanzado más allá del último elemento de la colección. ElementoActual desreferencia el objeto estado y devuelve el elemento de la colección al cual se refiere. Copiar devuelve una copia del objeto estado. Esto es útil para marcar un punto en una interacción.

Dada una clase TipoDeElemento, podemos iterar sobre una colección de instancias suyas como sigue [58]:

```
class TipoDeElemento {
public:
     void Procesar!);
     // …
};

Coleccion<TipoDeElemento*> unaColeccion;
EstadoIteracion* estado;

estado = unaColeccion.CrearEstadoInicial();

while (!unaColeccion.HaTerminado(estado)) {
     unaColeccion.ElementoActual(estado)-
>Procesar();
     unaColeccion.Siguiente(estado);
}
delete estado;
```

La interfaz de la iteración basada en el memento tiene dos beneficios interesantes:

1. Puede haber más de un estado para la misma colección (y lo mismo es cierto para el patrón Iterator (237)).
2. No necesita romper la encapsulación para permitir la iteración. El memento sólo es interpretado por la propia colección; nadie más tiene acceso a él. Otros enfoques para Iterar requieren romper la encapsulación haciendo a las clases iterador amigas de las clases de sus colecciones (véase el patrón Iterator (237)). La situación es a la inversa en la implementación basada en el memento: Colección es amiga de IteratorState.

El toolkit de resolución de problemas QOCA guarda información incremental en mementos [HHMV92]. Los clientes pueden obtener un memento que represente la solución actual a un sistema de ecuaciones. El memento contiene sólo aquellas variables de las ecuaciones que han cambiado desde la última solución. Normalmente, para cada nueva solución sólo cambia un pequeño subconjunto de las variables del resolvente. Este subconjunto es suficiente para devolver el resolvente a su solución precedente; volver a soluciones anteriores requiere almacenar mementos de las soluciones intermedias. Por tanto, no.se pueden establecer mementos en cualquier orden; QOCA se basa en un mecanismo de historial para revertir a soluciones anteriores.

PATRONES RELACIONADOS

Command (215); las órdenes pueden usar mementos para guardar el estado de las operaciones que pueden deshacerse.

Iterator (237): puede usar mentemos para la iteración, tal y como acabamos de describir.

OBSERVER
(Observador)

PROPÓSITO

Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que dependen de él.

TAMBIÉN CONOCIDO COMO

Dependents (Dependientes), *Publish-Subscribe* (Publicar-Suscribir)

MOTIVACIÓN

Un efecto lateral habitual de dividir un sistema en una colección de clases cooperantes es la necesidad de mantener una consistencia entre objetos relacionados. No queremos alcanzar esa consistencia haciendo a las clases fuertemente acopladas, ya que eso reduciría su reutilización.

Por ejemplo, muchos toolkits de interfaces gráficas de usuario separan los aspectos de presentación de la interfaz de usuario de los datos de aplicación subyacentes [KP88, LVC89, P+88, WGM88]. Las clases que definen los datos de las aplicaciones y las representaciones pueden reutilizarse de forma independiente. También pueden trabajar juntas. Un objeto hoja de cálculo y un gráfico de barras pueden representar la información contenida en el mismo objeto de dalos de aplicación usando diferentes representaciones. La hoja de cálculo y el gráfico de barras no se conocen entre sí, permitiendo así reutilizar sólo aquél que se necesite. Pero se *comportan* como si lo hicieran. Cuando el usuario cambia la información de la hoja de cálculo, la barra de herramientas refleja los cambios inmediatamente, y viceversa.

Este comportamiento implica que la hoja de cálculo y el gráfico de barras son dependientes del objeto de datos y, por tanto, se les debería notificar cualquier cambio en el estado de éste. Y no hay razón para limitar a dos el número de objetos dependientes; puede haber cualquier número de interfaces de usuario diferentes para los mismos datos.

El patrón Observer describe cómo establecer estas relaciones. Los principales objetos de este patrón son el sujeto y el observador. Un sujeto puede tener cualquier número de observadores dependientes de él. Cada vez que el sujeto cambia su estado se notifica a todos sus observadores. En respuesta, cada observador consultará al sujeto para sincronizar su estado con el estado de éste.

Este tipo de interacción también se conoce como publicar-suscribir. El sujeto es quien publica las notificaciones. Envía estas notificaciones sin tener que conocer quiénes son sus observadores. Pueden suscribirse un número indeterminado de observadores para recibir notificaciones.

APLICABILIDAD

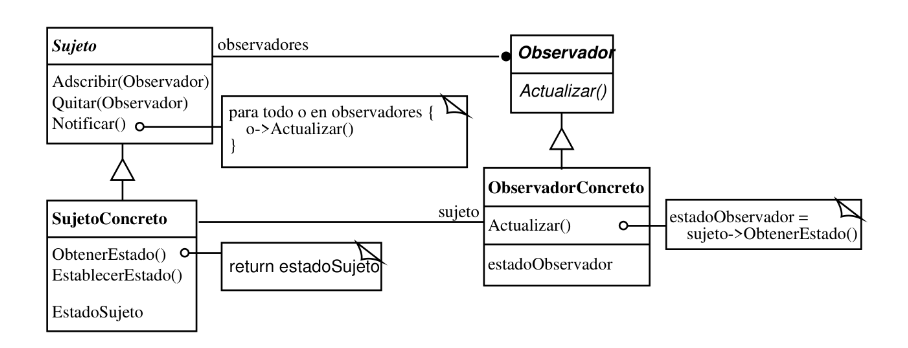
Úsese el patrón Observer en cualquiera de las situaciones siguientes:

- Cuando una abstracción tiene dos aspectos y uno depende del otro. Encapsular estos aspectos en objetos separados permite

modificarlos y reutilizarlos de forma independiente.

- Cuando un cambio en un objeto requiere cambiar otros, y no sabemos cuántos objetos necesitan cambiarse.
- Cuando un objeto debería ser capaz de notificar a otros sin hacer suposiciones sobre quiénes son dichos objetos. En otras palabras, cuando no queremos que estos objetos estén fuertemente acoplados.

ESTRUCTURA



PARTICIPANTES

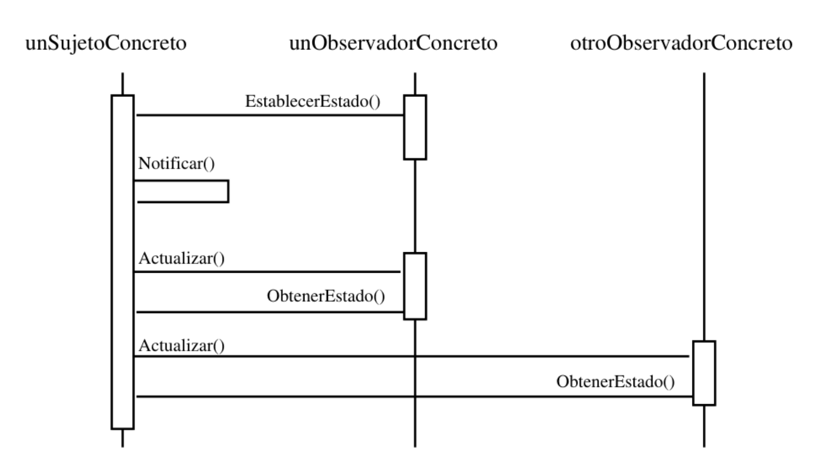
- **Sujeto**
  - conoce a sus observadores. Un sujeto puede ser observado por cualquier número de objetos Observador.
  - proporciona una interfaz para asignar y quitar objetos Observador.
- **Observador**
  - define una interfaz para actualizar los objetos que deben ser notificados ante cambios en un sujeto.
- **SujetoConcreto**
  - almacena el estado de interés para los objetos ObservadorConcreto.
  - envía una notificación a sus observadores cuando cambia su estado.
- **ObservadorConcreto**

- mantiene una referencia a un objeto SujetoConcreto.
- guarda un estado que debería ser consistente con el del sujeto.
- implementa la interfaz de actualización del Observador para mantener su estado consistente con el del sujeto.

COLABORACIONES

- SujetoConcreto notifica a sus observadores cada vez que se produce un cambio que pudiera hacer que el estado de éstos fuera inconsistente con el suyo.
- Después de ser informado de un cambio en el sujeto concreto, un objeto ObservadorConcreto puede pedirle al sujeto más información. ObservadorConcreto usa esta información para sincronizar su estado con el del sujeto.

El siguiente diagrama de interacción muestra las colaboraciones entre un sujeto y dos observadores:



Nótese cómo el objeto Observador que inicializa la petición de cambio pospone su actualización hasta que obtiene una notificación del sujeto. Notificar no siempre es llamado por el sujeto. Puede ser llamado por un observador o por un tipo de objeto completamente diferente. La sección de Implementación examina algunas variantes

comunes.

CONSECUENCIAS

El patrón Observador permite modificar los sujetos y observadores de forma independiente. Es posible reutilizar objetos sin reutilizar sus observadores, y viceversa. Esto permite añadir observadores sin modificar el sujeto u otros observadores.

Otras ventajas e inconvenientes del patrón Observer son los siguientes:

1. *Acoplamiento abstracto entre Sujeto y Observador*. Todo lo que un sujeto sabe es que tiene una lista de observadores, cada uno de los cuales se ajusta a la interfaz simple de la clase abstracta Observador. El sujeto no conoce la clase concreta de ningún observador. Por tanto el acoplamiento entre sujetos y observadores es mínimo.

   Gracias a que Sujeto y Observador no están fuertemente acoplados, pueden pertenecer a diferentes capas de abstracción de un sistema. Un sujeto de bajo nivel puede comunicarse e informar a un observador de más alto nivel, manteniendo de este modo intacta la estructura de capas del sistema. Si juntásemos al Sujeto y al Observador en un solo objeto, entonces el objeto resultante debería dividirse en dos capas (violando así la separación en capas) o estaría obligado a residir en una capa u otra (lo que puede comprometer la abstracción en capas).

2. *Capacidad de comunicación mediante difusión*. A diferencia de una petición ordinaria, la notificación enviada por un sujeto no necesita especificar su receptor. La notificación se envía automáticamente a todos los objetos interesados que se hayan suscrito a ella. Al sujeto no le importa cuántos objetos interesados haya; su única responsabilidad es notificar a sus observadores. Esto nos da la libertad de añadir y quitar observadores en cualquier momento. Se deja al observador manejar u obviar una notificación.

3. *Actualizaciones inesperadas*. Dado que los observadores no saben de la presencia de los otros, pueden no saber el coste último de cambiar el sujeto. Una operación aparentemente inofensiva sobre el sujeto puede dar lugar a una serie de actualizaciones en cascada de los observadores y sus objetos dependientes. Más aún, los criterios de dependencia que no están bien definidos o mantenidos suelen provocar falsas actualizaciones, que pueden ser muy difíciles de localizar,

   Este problema se ve agravado por el hecho de que el protocolo de actualización simple no proporciona detalles acerca de *qué* ha cambiado en el sujeto. Sin protocolos adicionales para ayudar a los observadores a descubrir qué ha cambiado, pueden verse obligados a trabajar duro para deducir los cambios.

IMPLEMENTACIÓN

En esta sección se examinan varias cuestiones relativas a la implementación del mecanismo de dependencia.

1. *Correspondencia entre los sujetos y sus observadores*. El modo más simple de que un sujeto conozca a los observadores a los que debería notificar es guardar referencias a ellos explícitamente en el sujeto. Sin embargo, dicho almacenamiento puede ser demasiado costoso cuando hay muchos sujetos y pocos observadores. Una solución es intercambiar espacio por tiempo usando una búsqueda asociativa (por ejemplo, mediante una tabla de dispersión —en inglés, tabla *hash*—) para mantener la correspondencia sujeto-observador. Así, un sujeto que no tenga observadores no incurrirá en ningún coste de almacenamiento. Por otro lado, este enfoque incrementa el coste de acceder a los observadores.

2. *Observar más de un sujeto*. Puede tener sentido en algunas situaciones que un observador dependa de más de un sujeto. Por ejemplo, una hoja de cálculo puede depender de más de un origen de datos. En tales casos es necesario

extender la interfaz de Actualizar para que el observador sepa *qué* sujeto está enviando la notificación. El sujeto puede simplemente pasarse a sí mismo como parámetro en la operación Actualizar, permitiendo así al observador saber qué sujeto examinar.

3. *¿Quién dispara la actualización*? El sujeto y sus observadores se basan en el mecanismo de notificación para permanecer consistentes. Pero ¿qué objeto llama realmente a Notificar para disparar la actualización? He aquí dos posibilidades:

- Hacer que las operaciones que establezcan el estado del Sujeto llamen a Notificar después de cambiar el estado del mismo. La ventaja de este enfoque es que los clientes no tienen que acordarse de llamar a Notificar sobre el sujeto. El inconveniente es que varias operaciones consecutivas provocarán varias actualizaciones consecutivas, lo que puede ser ineficiente.
- Hacer que los clientes sean responsables de llamar a Notificar en el momento adecuado. La ventaja aquí es que el cliente puede esperar a disparar la actualización hasta que se produzcan una serie de cambios de estado, evitando así las innecesarias actualizaciones intermedias. El inconveniente es que los clientes tienen la responsabilidad añadida de disparar la actualización. Eso hace que sea propenso a errores, ya que los clientes pueden olvidarse de-llamar a Notificar.

4. *Referencias perdidas a los sujetas borrados*. Borrar un sujeto no debería producir referencias perdidas en sus observadores. Una manera de evitar esto es hacer que el sujeto notifique a sus observadores cuando va a ser borrado, para que éstos puedan inicializar la referencia al sujeto. En general, borrar los observadores no suele ser una opción, ya que puede haber otros objetos que hagan referencia a ellos, y también pueden estar observando a otros sujetos.

5. *Asegurarse de que el estada del Sujeto es consistente*

consigo mismo antes de la notificación. Es importante garantizar que el estado del Sujeto es consistente consigo mismo antes de llamar a Notificar, porque los observadores le piden al sujeto su estado actual mientras actualizan su propio estado.

Es fácil violar involuntariamente esta regla de auto-consistencia cuando las operaciones de las subclases de Sujeto llaman a operaciones heredadas. Por ejemplo, en la siguiente secuencia de código la actualización se dispara cuando el sujeto se encuentra en un estado inconsistente:

```
void     MiSujeto::Operacion     (int
nuevoValor) {
   ClaseBaseSujeto::Operacion(nuevoValor);
      // se dispara la notificación

   _miVar += nuevoValor;
      // se actualiza el estado de la
subclase
      // (¡demasiado tarde!)
}
```

Se puede salvar este escollo enviando notificaciones en métodos plantilla (Template Method (299)) de la clase abstracta Sujeto, definiendo una operación primitiva para que sea redefinida por las subclases y haciendo que Notificar sea la última operación del método plantilla, lo que garantizará que el objeto es consistente consigo mismo cuando las subclases redefinan las operaciones de Sujeto.

```
void Texto::Cortar (SeleccionDeTexto
t) {
     SustituirSeleccion(t);        //
redefinida en las subclases
     Notificar();
}
```

Por cierto, siempre es una buena idea documentar qué

operaciones del Sujeto disparan notificaciones.

6. *Evitar protocolos específicos del observador: los modelos push y pull*. Las implementaciones del patrón Observer suelen hacer que el sujeto envíe información adicional sobre el cambio. El sujeto pasa esta información como un parámetro de Actualizar. La cantidad de información puede variar mucho.

   En un extremo, al que llamaremos modelo *push*, el sujeto envía a los observadores información detallada acerca del cambio, ya quieran éstos o no. El otro extremo es el modelo *pulí]* el sujeto no envía nada más que la notificación mínima, y los observadores piden después los detalles explícitamente.

   El modelo pull enfatiza la ignorancia del sujeto respecto a sus observadores, mientras que el modelo push asume que los sujetos saben algo sobre las necesidades de sus observadores. El modelo push puede hacer que los observadores sean menos reutilizables, ya que las clases Sujeto hacen suposiciones sobre las clases Observador que pudieran no ser siempre ciertas. Por otro lado, el modelo pull puede ser ineficiente, ya que las clases Observador deben determinar qué ha cambiado sin ayuda por parte del Sujeto.

7. *Especificar las modificaciones de interés explícitamente*. Se puede mejorar la eficiencia extendiendo la interfaz de registro del sujeto para permitir que los observadores registren sólo a aquellos eventos concretos que les interesen. Cuando ocurre uno de tales eventos, el sujeto informa únicamente a aquellos observadores que se han registrados como interesados en ese evento. Una manera de permitir esto es usar la noción de aspectos en los objetos Sujeto. Para registrarse como interesado en eventos particulares, los observadores se adscriben a sus sujetos usando

```
void
```

```
Sujeto::Adscribir(Observador*,
Aspecto& interes);
```

donde interes especifica el evento de interés. En el momento de la notificación, el sujeto proporciona a sus observadores el aspecto que ha cambiado como un parámetro de la operación Actualizar. Por ejemplo:

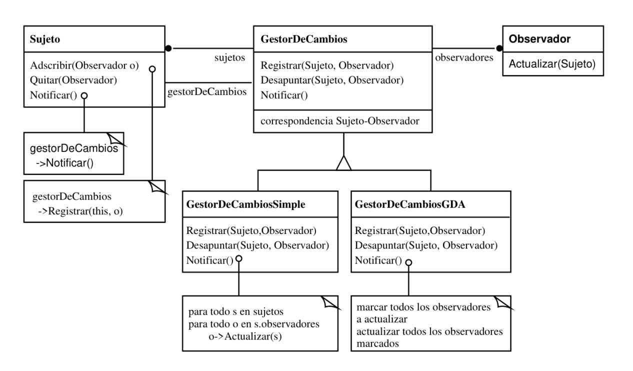
```
void
   Observador::Actualizar(Sujeto*,
   Aspecto& interes);
```

8. *Encapsular la semántica de las actualizaciones complejas*. Cuando la relación de dependencia entre sujetos y observadores es particularmente compleja, puede ser necesario un objeto que mantenga estas relaciones. Llamaremos a este objeto un GestorDeCambios. Su propósito es minimizar el trabajo necesario para lograr que los observadores reflejen un cambio en su sujeto. Por ejemplo, si una operación necesita cambiar varios sujetos interdependientes, puede ser necesario asegurarse de que se notifica a sus observadores sólo después de que *todos* los sujetos han sido modificados, para evitar notificar a los observadores más de una vez. GestorDeCambios tiene tres responsabilidades:

- Hace corresponder a un sujeto con sus observadores, proporcionando una interfaz para mantener dicha correspondencia. Esto elimina la necesidad de que los sujetos mantengan referencias a otros observadores y viceversa.
- Define una determinada estrategia de actualización.
- Actualiza todos los observadores dependientes a petición de un sujeto.

El diagrama siguiente representa una implementación del patrón Observer basada en un GestorDeCambios simple. Hay dos tipos de GestorDeCambios. GestorDeCambiosSimple es simplista en el sentido de que siempre actualiza todos los observadores de cada sujeto. Por el contrario, GestorDeCambiosGDA maneja grafos dirigidos-acíclicos de dependencias entre sujetos y sus

observadores. Es preferible un GestorDeCambiosGDA frente a un GestorDeCambiosSimple cuando un observador observa a más de un sujeto. En ese caso, un cambio en dos o más sujetos podría causar actualizaciones redundantes. El GestorDeCambiosGDA garantiza que el observador sólo recibe una única actualización. GestorDeCambiosSimple está bien cuando las actualizaciones múltiples no constituyen un problema.



GestorDeCambios es una instancia del patrón Mediator (251). En general, sólo hay un único GestorDeCambios, y es conocido globalmente. El patrón Singleton (119) sería aquí de utilidad.

9. *Unir las clases Sujeto y Observador*. Las bibliotecas de clases escritas en lenguajes que carecen de herencia múltiple (como Smalltalk) generalmente no definen clases separadas Sujeto y Observador, sino que juntan sus interfaces en una clase. Eso permite definir un objeto que haga tanto de sujeto como de observador sin usar herencia múltiple. En Smalltalk, por ejemplo, las interfaces de Sujeto y Observador se definen en la clase raíz Object, haciéndolas así disponibles para todas las clases.

CÓDIGO DE EJEMPLO

Una clase abstracta define la interfaz Observador:

```
class Sujeto;
```

```
class Observador {
public:
     virtual ~ Observador();
     virtual     void     Actualizar(Sujeto*
elSujetoQueCambio) = 0;
protected:
     Observador();
};
```

Esta implementación permite múltiples sujetos por cada observador. El sujeto que se pasa a la operación Actualizar permite que el observador determine qué objeto ha cambiado cuando éste observa más de uno.

De forma similar, una clase abstracta define la interfaz de Sujeto:

```
class Sujeto {
public:
     virtual ~Sujeto();

     virtual, void Adscribir(Observador*);
     virtual void Quitar(Observador*);
     virtual void Notificar();
protected:
     Sujeto();
private:
     lista<Observador*> *_observadores;
};

void Sujeto::Adscribir (Observador* o) {
     _observadores->Insertar(o);
}

void Sujeto::Quitar (Observador* o) {
     _observadores->Eliminar(o);
}

void Sujeto::Notificar () {
     Iteradorlista<Observador*>
i(_observadores);

     for    (i.Primero));    !i.HaTerminado();
i.Siguiente()) {
          i.ElementoActual()-
>Actualizar(this);
```

```
        }
    }
```

Reloj es un sujeto concreto que almacena y mantiene la hora del día, notificando a sus observadores cada segundo. Reloj proporciona la interfaz para obtener unidades de tiempo por separado, como la hora, los mininos o los segundos.

```
class Reloj : public Sujeto {
public:
    Reloj();

    virtual int ObtenerHora();
    virtual int ObtenerMinuto();
    virtual int ObtenerSegundo();

    void Pulso();
};
```

La operación Pulso es llamada por un reloj interno a intervalos de tiempo regulares para proporcionar una base de tiempo fiable. Pulso actualiza el estado interno de Reloj y llama a Notificar para informar a los observadores del cambio:

```
void Reloj::Pulso () {
    //  actualiza  el  estado  del  tiempo
interno
    // …
    Notificar() ;
}
```

Ahora podemos definir una clase RelojDigital que muestra el tiempo. Esta clase hereda su funcionalidad gráfica de una clase Util **[59]** proporcionada por un toolkit de interfaces de usuario. La interfaz del Observador se combina con la de Reloj Digital heredando de Observador.

```
class  RelojDigital:  public  Util,  public
```

```
Observador {
public:
     RelojDigital(Reloj *);
     virtual ~RelojDigital();

     virtual void Actualizar(Sujeto*);
          //   redefine   la   operación   do
Observador

     virtual void Dibujar();
          // redefine la operación de Util;
          //  define  cómo  dibujar  el  reloj
digital

private:
     Reloj* _sujeto;
};

RelojDigital::RelojDigital (Reloj* s) {
     _sujeto = s;
     _sujeto->Adscribir(this);
}

RelojDigital:: RelojDigital () {
     _sujeto->Quitar(this);
}
```

Antes de que la operación Actualizar dibuje la apariencia del reloj, se comprueba que el sujeto de la notificación sea el sujeto del reloj:

```
void    RelojDigital::Actualizar    (Sujeto*
elSujetoQueCambio) {
     if (elSujetoQueCambio == _sujeto) {
          dibujar();
     }
}

void RelojDigital::Dibujar () {
     // obtiene los nuevos valores del sujeto

     int hora = _sujeto->ObtenerHora();
     int minuto = _sujeto->ObtenerMinuto();
     // etc.
     // dibuja el reloj digital
}
```

Se puede definir una clase RelojAnalogico de la misma manera.

```
class RelojAnalogico : public Util, public
Observador {
public:
     RelojAnalogico(Reloj*);
     virtual void Actualizar(Sujeto*);
     virtual void Dibujar();
     // …
};
```

El siguiente código crea un RelojAnalogico y un RelojDigital que siempre muestra el mismo tiempo:

```
Reloj* reloj = new Reloj;
RelojAnalogico*    relojAnalogico    =    new
RelojAnalogico(reloj);
RelojDigital*     relojDigital     =     new
RelojDigital(reloj);
```

Cada vez que el reloj emite un pulso, los dos relojes se actualizarán y se volverán a dibujar de manera apropiada.

USOS CONOCIDOS

El primer y tal vez más importante ejemplo de patrón Observer aparece en el Modelo/Vista/Controlador de Smalltalk (MVC), el framework de interfaces de usuario en el entorno de Smalltalk [KP88J. La clase Modelo en MCV desempeña el papel de Sujeto, mientras que Vista es la clase base de los observadores. Smalltalk, ET++ [WGM88] y la biblioteca de clases THINK [Sym93b] proporcionan un mecanismo general de dependencia poniendo las interfaces de Sujeto y Observador en la clase base de todas las otras clases del sistema.

Otros toolkits de interfaces de usuario que emplean este patrón son Interviews [LVC89J, Andrew Toolkit [P+88] y Unidraw [VL90]. Interviews define explícitamente las clases Observer y Observable (para los sujetos). Andrew las llama “vista” y “objeto de

datos", respectivamente. Unidraw divide los objetos del editor gráfico en partes Vista (para los observadores) y Sujeto.

PATRONES RELACIONADOS

Mediator (251): encapsulando semánticas de actualizaciones complejas, el GestorDeCambios actúa como mediador entre sujetos y observadores.

Singleton (119): el GestorDeCambios puede usar el patrón Singleton para que sea único y globalmente accesible.

STATE
(Estado)

PROPÓSITO

Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.

TAMBIÉN CONOCIDO COMO

Objects for States (Estados como Objetos)

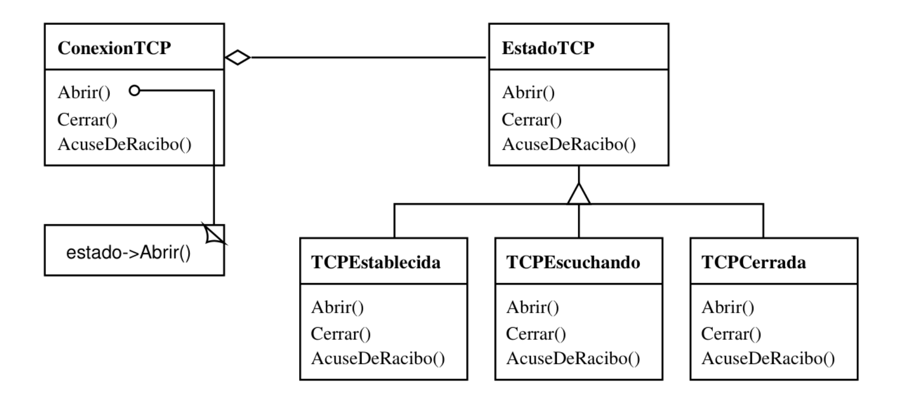
MOTIVACIÓN

Pensemos en una clase ConexionTCP que representa una conexión de red. Un objeto ConexionTCP puede encontrarse en uno de los siguientes estados: Establecida, Escuchando o Cerrada. Cuando un objeto ConexionTCP recibe peticiones de otros objetos, les responde de distinta forma dependiendo de su estado actual. Por ejemplo, el efecto de una petición Abrir depende de si la conexión se encuentra en su estado Cerrada o en su estado Establecida. El patrón State describe cómo puede ConexionTCP exhibir un comportamiento diferente en cada estado.

La idea clave de este patrón es introducir una clase abstracta llamada EstadoTCP que representa los estados de la conexión de red. La clase EstadoTCP declara una interfaz común para todas las clases que representan diferentes estados operacionales. Las subclases de EstadoTCP implementan comportamiento específico de cada estado. Por ejemplo, las clases TCPEstablecida y TCPCerrada implementan comportamiento concreto de los estados Establecida y Cerrada de una ConexionTCP.

La clase ConexionTCP mantiene un objeto de estado (una instancia de una subclase de EstadoTCP) que representa el estado actual de la conexión TCP. La clase ConexionTCP delega todas las peticiones dependientes del estado en este objeto de estado.

ConexionTCP usa su instancia de la subclase de EstadoTCP para realizar operaciones que dependen del estado de la conexión.



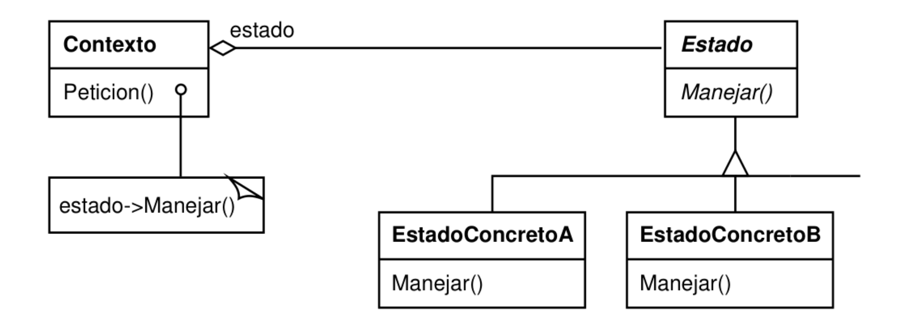
Cada vez que cambia el estado de la conexión, el objeto ConexionTCP cambia el objeto de estado que usa. Cuando la conexión pasa de establecida a cerrada, por ejemplo, ConexionTCP sustituirá su instancia de TCPEstablccida por una instancia de TCPCerrada.

APLICABILIDAD

Úsese el patrón State en cualquiera de los siguientes dos casos:

- El comportamiento de un objeto depende de su estado, y debe cambiar en tiempo de ejecución dependiendo de ese estado.
- Las operaciones tienen largas sentencias condicionales con múltiples ramas que dependen del estado del objeto. Este estado se suele representar por una o más constantes enumeradas. Muchas veces son varias las operaciones que contienen esta misma estructura-condicional. El patrón State pone cada rama de la condición en una clase aparte. Esto nos permite tratar al estado del objeto como un objeto de pleno derecho que puede variar independientemente de otros objetos.

ESTRUCTURA



PARTICIPANTES

- **Contexto** (ConexionTCP)
  - define la interfaz de interés para los clientes.
  - mantiene una instancia de una subclase de EstadoConcreto que define el estado actual.
- **Estado** (EstadoTCP)
  - define una interfaz para encapsular el comportamiento asociado con un determinado estado del Contexto.
- **subclases de EstadoConcreto** (TCPEstablecida, TCPEscuchando, TCPCerrada)
  - cada subclase implemento un comportamiento asociado con un estado del Contexto.

COLABORACIONES

- Contexto delega las peticiones que dependen del estado en el objeto EstadoConcreto actual.
- Un contexto puede pasarse a sí mismo como parámetro para que el objeto Estado maneje la petición. Esto permite al objeto Estado acceder al contexto si fuera necesario.
- Contexto es la interfaz principal para los clientes. Los clientes pueden configurar un contexto con objetos Estado. Una vez que está configurado el contexto, sus clientes ya no tienen que tratar con los objetos Estado directamente.
- Cualquiera de las subclases de Contexto o de EstadoConcreto pueden decidir qué estado sigue a otro y bajo qué

circunstancias.

CONSECUENCIAS

El patrón State tiene las siguientes consecuencias:

1. *Localiza el comportamiento dependiente del estado y divide dicho comportamiento en diferentes estados*. El patrón State sitúa en un objeto todo el comportamiento asociado con un determinado estado. Como todo el código dependiente del estado reside en una subclase de Estado, pueden añadirse fácilmente nuevos estados y transiciones definiendo nuevas subclases. Una alternativa es usar valores de datos para definir los estados internos y hacer que las operaciones de Contexto comprueben dichos datos explícitamente. Pero en ese caso tendríamos sentencias condicionales repartidas por toda la implementación de Contexto. Añadir un nuevo estado podría requerir cambiar varias operaciones, complicando el mantenimiento.

   El patrón State evita este problema, pero puede introducir otro, al distribuir el comportamiento para los diferentes estados en varias subclases de Estado. Esto incrementa el número de clases y es menos compacto que una única clase. Pero dicha distribución es realmente buena si hay muchos estados, que de otro modo necesitarían grandes sentencias condicionales.

   Al igual que ocurre con los procedimientos largos, hay que tratar de evitar las grandes sentencias condicionales. Son monolíticas y tienden a hacer el código menos explícito, lo que a su vez las hace difíciles de modificar y extender. El patrón State ofrece un modo mejor de estructurar el código dependiente del estado. La lógica que determina las transiciones entre estados no reside en sentencias if o switch monolíticas, sino que se reparte entre las subclases de Estado. Al encapsular cada transición y acción en una clase estamos elevando la idea de un estado de ejecución a objetos de estado en toda regla. Esto impone una estructura al código y hace que su intención sea más clara.

2. *Hace explícitas las transiciones entre estados*. Cuando un objeto define su estado actual únicamente en términos de valores de datos internos, sus transiciones entre estados carecen de una representación explícita; sólo aparecen como asignaciones a determinadas variables. Introducir objetos separados para los diferentes estados hace que las transiciones sean más explícitas. Además, los objetos Estado pueden proteger al Contexto frente a estados internos inconsistentes, ya que las transiciones entre estados son atómicas desde una perspectiva del Contexto —tienen lugar cambiando *una* variable (el objeto variable del Contexto, Estado), no varias [dCLF93]—.

3. *Los objetos Estado pueden compartirse*. En caso de que los objetos Estado no tengan variables —es decir, si el estado que representan está totalmente representado por su tipo — entonces varios contextos pueden compartir un mismo objeto Estado. Cuando se comparten los estados de este modo, son en esencia pesos ligeros (*véase* el patrón Flyweight (179)) que no tienen estado intrínseco, sino sólo comportamiento.

IMPLEMENTACIÓN

El patrón Estado da lugar a una serie de cuestiones de implementación:

1. *¿Quién define las transiciones entre estados*? El patrón State no especifica qué participante define los criterios para las transiciones entre estados. Si estos criterios son fijos, entonces pueden implementarse enteramente en el Contexto. No obstante, es generalmente más flexible y conveniente que sean las propias subclases de Estado quienes especifiquen su estado sucesor y cuándo llevar a cabo la transición. Esto requiere añadir una interfaz al Contexto que permita a los objetos Estado asignar explícitamente el estado actual del Contexto. Descentralizar de esta forma la lógica de transición facilita modificar o extender dicha lógica definiendo nuevas

subclases de Estado. Una desventaja de la descentralización es que una subclase de Estado conocerá al menos a otra, lo que introduce dependencias de implementación entre subclases.

2. *Una alternativa basada en tablas*. Cargill, en su libro *C++ Programming Style* [Car92], describe otra forma de estructurar el código dirigido por estados: mediante tablas que hagan corresponder entradas con transiciones de estado. Para cada estado, una tabla hace corresponder cada posible entrada con un estado sucesor. Este enfoque convierte código condicional (y funciones virtuales, en el caso del patrón State) en una tabla de búsqueda.

   La principal ventaja de las tablas es su regularidad: se pueden cambiar los criterios de transición modificando datos en vez del código de programa. Hay, no obstante, algunos inconvenientes:

- Una tabla de búsqueda es normalmente menos eficiente que una llamada a una función (virtual).
- Situar la lógica de transición en un formato tabular uniforme hace que los criterios de transición sean menos explícitos y, por tanto, más difíciles de comprender.
- Suele dificultar añadir acciones que acompañen a las transiciones de estado. El enfoque dirigido por una tabla representa los estados y sus transiciones, pero debe ser aumentado para realizar algún tipo de procesamiento arbitrario con cada transición.

   Las principales diferencias entre las máquinas de estados basadas en tablas y el patrón State se pueden resumir en ésta: el patrón State modela comportamiento específico del estado, mientras que el enfoque basado en una tabla se centra en definir las transiciones de estado.

3. *Crear y destruir objetos Estado*. Una cuestión de implementación que hay que ponderar es si (1) crear los objetos Estado sólo cuando se necesitan y destruirlos después, o si (2) crearlos al principio y no destruirlos

nunca.

La primera elección es preferible cuando no se conocen los estados en tiempo de ejecución y los contextos cambian de estado con poca frecuencia. Este enfoque evita crear objetos que no se usarán nunca, lo que puede ser importante si los objetos Estado guardan una gran cantidad de información. El segundo enfoque es mejor cuando los cambios tienen lugar rápidamente, en cuyo caso querremos evitar destruir los estados, ya que pueden volver a necesitarse de nuevo en breve. Los costes de creación se pagan una vez al principio, y no existen costes de destrucción. No obstante, este enfoque puede no ser apropiado, ya que el Contexto debe guardar referencias a todos los estados en los que pudiera entrar.

4. *Usar herencia dinámica*. Cambiar el comportamiento de una determinada petición podría lograrse cambiando la clase del objeto en tiempo de ejecución, pero esto no es posible en la mayor parte de los lenguajes de programación orientados a objetos. Las excepciones incluyen Self [US87] y otros lenguajes basados en delegación que proporcionan dicho mecanismo y por tanto admiten el patrón State directamente. Los objetos en Self pueden delegar operaciones en otros objetos para obtener una especie de herencia dinámica. Cambiar el destino de la delegación en tiempo de ejecución cambia por tanto la estructura de herencia. Este mecanismo permite que los objetos cambien su comportamiento, y viene a ser lo mismo que cambiar su clase.

CÓDIGO DE EJEMPLO

El ejemplo siguiente muestra el código C++ para el ejemplo de la conexión TCP que se describió en la sección de Motivación. Este ejemplo es una versión simplificada del protocolo TCP; no se describe el protocolo completo ni todos los estados de las conexiones TCP [60].

En primer lugar, definimos la clase ConexionTCP, que proporciona una interfaz para transmitir datos y que procesa las

peticiones para cambiar el estado.

```
class FlujoOctetosTCP;
class EstadoTCP;

class ConexionTCP {
public:
   ConexionTCP();

   void AbrirActiva();
   void AbrirPasiva();
   void Cerrar();

   void Enviar();
   void AcuseDeRecibo();
   void Sincronizar();
   void ProcesarOcteto(FlujoOctetosTCP*);
prívate:
   friend class EstadoTCP;
   void CambiarEstado(EstadoTCP*);
private:
   EstadoTCP* _estado;
};
```

ConexionTCP guarda una instancia de la clase EstadoTCP en la variable miembro _estado. La clase EstadoTCP duplica la interfaz para cambiar el estado de ConexionTCP. Cada operación EstadoTCP recibe como parámetro una instancia de ConexionTCP, lo que permite a EstadoTCP acceder a los datos de ConexionTCP y cambiar el estado de la conexión.

```
class EstadoTCP {
public:
   virtual    void    Transmitir(ConexionTCP*,
FlujoOctetosTCP*);
   virtual void AbrirActiva(ConexionTCP*);
   virtual void AbrirPasiva(ConexionTCP*);
   virtual void Cerrar(ConexionTCP*);
   virtual void Sincronizar(ConexionTCP*);
   virtual void AcuseDeRecibo(ConexionTCP*);
   virtual void Enviar(ConexionTCP*);
protected:
   void           CambiarEstado(ConexionTCP*,
```

```
EstadoTCP*);
};
```

ConexionTCP delega todas las peticiones dependientes del estado en su instancia de EstadoTCP, _estado. ConexionTCP también proporciona una operación para cambiar esta variable por un nuevo EstadoTCP. El constructor de ConexionTCP inicializa este objeto al estado TCPCerrada (definida más tarde),

```
ConexionTCP::ConexionTCP () {
   _estado = TCPCerrada::Instancia();
}

void  ConexionTCP::CambiarEstado  (EstadoTCP*
e) {
   _estado = e;
}

void ConexionTCP::AbrirActiva () {
   _estado->AbrirActiva(this);
}

void ConexionTCP::AbrirPasiva () {
   estado->AbrirPasiva(this);
}

void ConexionTCP::Cerrar () {
   _estado->Cerrar(this);
}

void ConexionTCP::AcuseDeRecibo () {
   _estado->AcuseDeRecibo(this);
}

void ConexionTCP::Sincronizar () {
   estado->Sincronizar(this);
}
```

EstadoTCP implementa el comportamiento predeterminado de todas las peticiones delegadas en él. También puede cambiar el estado de una ConexionTCP mediante la operación CambiarEstado. EstadoTCP se declara como amiga de ConexionTCP para dar a esta operación un acceso restringido.

```
void   EstadoTCP::Transmitir   (ConexionTCP*,
FlujoOctetosTCP*) { }
void EstadoTCP::AbrirActiva (ConexionTCP*) {
}
void EstadoTCP::AbrirPasiva (ConexionTCP*) {
}
void EstadoTCP::Cerrar (ConexionTCP*) { }
void EstadoTCP::Sincronizar (ConexionTCP*) {
}

void  EstadoTCP::CambiarEstado  (ConexionTCP*
c, EstadoTCP* e) {
   c->CambiarEstado(e);
}
```

Las subclases de EstadoTCP implementan comportamiento específico de ese estado. Una conexión TCP puede encontrarse en muchos estados: Establecida, Escuchando, Cerrada, etc., y hay una subclase de EstadoTCP para cada uno de ellos. Examinaremos estas bes subclases en detalle: TCPEstablecida, TCPEscuchando y TCPCerrada.

```
class TCPEstablecida : public EstadoTCP {
public:
   static EstadoTCP* Instancia();

   virtual    void    Transmitir(ConexionTCP*,
FlujoOctetosTCP*);
   virtual void Cerrar(ConexionTCP*);
};

class TCPEscuchando : public EstadoTCP {
public:
   static EstadoTCP* Instancia();

   virtual void Enviar(ConexionTCP*);
   // …
};

class TCPCerrada : public EstadoTCP {
public:
   static EstadoTCP* Instancia();

   virtual void AbrirActiva(ConexionTCP*);
   virtual void AbrirPasiva(ConexionTCP*);
```

```
   // …
};
```

Las subclases de EstadoTCP no mantienen un estado local, de modo que pueden compartirse, siendo sólo necesaria una instancia de cada una. Esta única instancia de cada subclase de EstadoTCP se obtiene a través de la operación estática Instancia **[61]**.

Cada subclase de EstadoTCP implementa comportamiento específico de ese estado para aquellas peticionéis válidas en este estado:

```
void TCPCerrada::AbrirActiva (ConexionTCP* c)
{
   // envía SYN, recibe SYN, ACK, etc.

   CambiarEstado(c,          TCPEstablecida::
Instancia());
}

void TCPCerrada::AbrirPasiva (ConexionTCP* c)
{
   CambiarEstado(c,
TCPEscuchando::Instancia());
}
void TCPEstablecida::Cerrar (ConexionTCP* c)
{
   // envía FIN, recibe ACK de FIN

   CambiarEstado(c,
TCPEscuchando::Instancia!));
}

void TCPEstablecida::Transmitir (
   ConexionTCP* c, FlujoOctetosTCP* o
) {
   c>ProcesarOcteto(o);
}

void TCPEscuchando::Enviar (ConexionTCP* c) {
   // envía SYN, recibe SYN, ACK, etc.

   CambiarEstado(c,
TCPEstablecida::Instancia());
}
```

Después de realizar el trabajo concreto de este estado, estas operaciones llaman a la operación CambiarEstado para cambiar el estado de ConexionTCP. ConexionTCP no sabe nada sobre el protocolo de conexión TCP; son las subclases de EstadoTCP quienes definen cada transición de estado y acción de TCP.

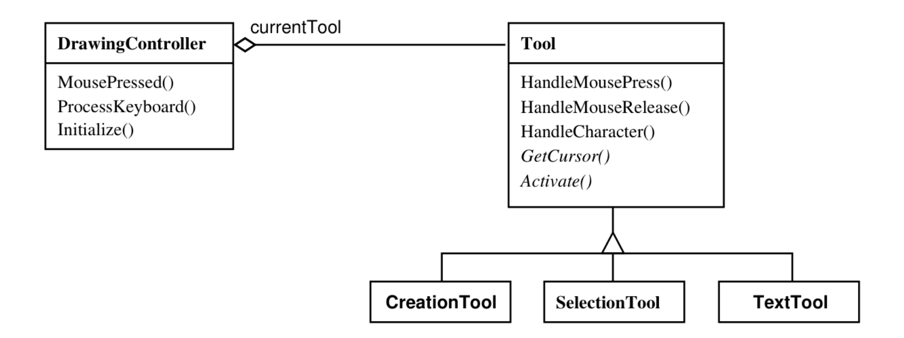
USOS CONOCIDOS

Johnson y Zweig [JZ91] describen el patrón State y su aplicación a los protocolos de conexión TCP.

La mayoría de programas de dibujo interactivos proporcionan “herramientas” para llevar a cabo operaciones mediante manipulación directa. Por ejemplo, una herramienta de dibujado de líneas permite al usuario hacer clic en un punto y arrastrar para crear una nueva línea. Una herramienta de selección permite al usuario seleccionar formas. Suele haber una paleta de herramientas para elegir. El usuario concibe esta actividad como tomar una herramienta y manejarla, pero en realidad el comportamiento del editor cambia con la herramienta actual: cuando está activa una herramienta de dibujado creamos formas; cuando está activa la herramienta de selección seleccionamos formas; y así sucesivamente. Podemos usar el patrón State para cambiar el comportamiento del editor que depende de la herramienta actual.

Podemos definir una clase abstracta Herramienta a partir de la cual definir subclases que implementen comportamiento específico de la herramienta. El editor de dibujo mantiene un objeto Herramienta actual al cual delega las peticiones. Cuando el usuario selecciona una nueva herramienta, cambia este objeto, haciendo que el comportamiento del editor cambie en consecuencia.

Esta técnica se emplea en los frameworks de editores de dibujo HotDraw [Joh92] y Unidraw [VL90]. Esto permite que los clientes definan nuevos tipos de herramientas fácilmente. En HotDraw, la clase DrawingController redirige las peticiones al objeto Tool (*herramienta*) actual. En Unidraw, las clases correspondientes son Viewer y Tool. El siguiente diagrama de clases muestra un esbozo de las interfaces Tool y DrawingController:



El modismo de Coplien Sobre-Carta (*Envelope-Letter*) [Cop92j está relacionado con el patrón State. Sobre-Carta es una técnica para cambiar la clase de un objeto en tiempo de ejecución. El patrón

State es más específico, centrándose en cómo tratar con un objeto cuyo comportamiento depende de su estado.

PATRONES RELACIONADOS

El patrón Flyweight (179) explica cuándo y cómo compartir objetos Estado. Los objetos Estado muchas veces son Singletons (119).

STRATEGY (Estrategia)

PROPÓSITO

Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.

TAMBIÉN CONOCIDO COMO

Policy (Política)

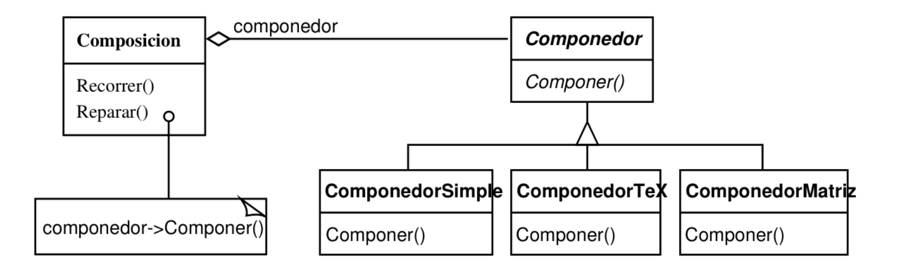
MOTIVACIÓN

Existen muchos algoritmos para dividir en líneas un flujo de texto. Codificar dichos algoritmos en las clases que los usan no resulta una buena práctica por varias razones:

Los clientes que necesitan dividir el texto en líneas se vuelven más complejos si tienen que incluir dicho código, lo que los hace más grandes y más difíciles de mantener, sobre todo si permiten varios algoritmos diferentes de división en líneas.

- Los distintos algoritmos serán apropiados en distintos momentos. No tenemos porqué permitir
- múltiples algoritmos si no los vamos a usar todos.
- Es difícil añadir nuevos algoritmos o modificar los existentes cuando la división en líneas es parte integral de un cliente.

Estos problemas pueden evitarse definiendo clases que encapsulen los diferentes algoritmos de división en líneas. Un algoritmo así encapsulado se denomina una estrategia.



Supongamos que una clase Composición debe mantener y actualizar los saltos de línea del texto mostrado en un visor. Las estrategias de división en líneas no están implementadas en la clase Composición. En vez de eso, se implementan separadamente por las subclases de la clase abstracta Componedor. Las subclases de Componedor implementan diferentes estrategias:

- ComponedorSimple implementa una estrategia simple que calcula un salto de línea cada vez.
- ComponedorTeX implements el algoritmo TeX para buscar los saltos de línea. Esta estrategia trata de optimizar los saltos de línea globalinente, es decir, un párrafo cada vez.
- ComponedorMatriz implements una estrategia que selecciona los saltos de línea de modo que cada fila tenga up número determinado de elementos. Es útil para dividir una serie de iconos en filas, por ejemplo.

Una Composición mantiene una referencia a un objeto Componedor. Cada vez que una Composición vuelve a formatear su texto, reenvía esta responsabilidad a su objeto Componedor. El cliente de Composición especifica qué Componedor debería usarse, y dicho Componedor será instalado en la Composición.

APLICABILIDAD

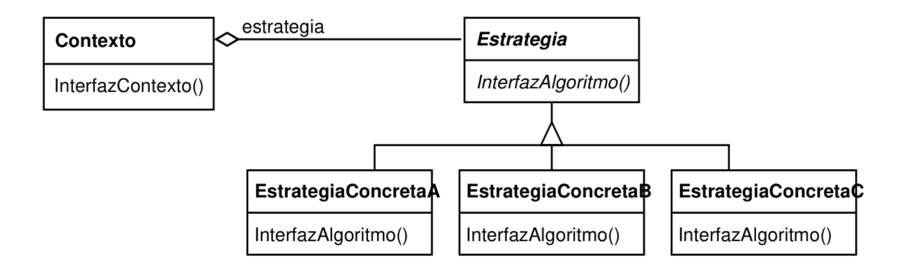
Úsese el patrón Strategy cuando

- muchas clases relacionadas difieren sólo en su comportamiento. Las estrategias permiten configurar una clase con un determinado comportamiento de entre muchos posibles.
- se necesitan distintas variantes de un algoritmo, for ejemplo,

podríamos definir algoritmos que reflejasen distintas soluciones de compromiso entre tiempo y espacio. Pueden usarse estrategias cuando estas variantes se implementan como una jerarquía de clases de algoritmos [HOS7],

- un algoritmo usa datos que los clientes no deberían conocer. Úsese el patrón Strategy para evitar exponer estructuras de dalos complejas y dependientes del algoritmo.
- una clase define muchos comportamientos, y éstos se representan como múltiples sentencias condicionales en sus operaciones. En vez de tener muchos condicionales, podemos mover las ramas de éstos a su propia clase Estrategia.

ESTRUCTURA



PARTICIPANTES

- **Estrategia** (Componedor)
  - declara una interfaz común a todos los algoritmos permitidos. El Contexto usa esta interfaz para llamar al algoritmo definido por una EstrategiaConcreta.
- **EstrategiaConcreta** (ComponedorSimple, ComponedorTeX, ComponedorMatriz)
  - implementa el algoritmo usando la interfaz Estrategia.
- **Contexto** (Composición)
  - se configura con un objeto EstrategiaConcreta.
  - mantiene una referencia a un objeto Estrategia.
  - puede definir una interfaz que permita a la Estrategia acceder a sus datos.

COLABORACIONES

- Estrategia y Contexto interactúan para implementar el algoritmo elegido. Un contexto puede pasar n la estrategia todos los datos requeridos por el algoritmo cada vez que se llama a éste. Otra alternativa es que el contexto se pase a sí mismo como argumento de las operaciones de Estrategia. Eso permite a la estrategia hacer llamadas al contexto cuando sea necesario.
- Un contexto redirige peticiones de los clientes a su estrategia. Los clientes normalmente crean un objeto EstrategiaConcreta, el cual pasan al contexto; por tanto, los clientes interactúan exclusivamente con el contexto. Suele haber una familia de clases EstrategiaConcreta a elegir por el cliente.

CONSECUENCIAS

El patrón Strategy presenta las siguientes ventajas e inconvenientes:

1. *Familias de algoritmos relacionados*. Las jerarquías de clases Estrategia definen una familia de algoritmos o comportamientos para ser reutilizados por los contextos. La herencia puede ayudar a sacar factor común: de la funcionalidad de estos algoritmos.

2. *Una alternativa a la herencia*. La herencia ofrece otra forma de permitir una variedad de algoritmos o comportamientos. Se puede heredar directamente de una clase Contexto para proporcionar diferentes comportamientos. Pero esto liga el comportamiento al Contexto, mezclando la implementación del algoritmo con la del Contexto, lo que hace que éste sea más difícil de comprender, mantener y extender. Y no se puede modificar el algoritmo dinámicamente. Acabaremos teniendo muchas clases relacionadas cuya única diferencia es el algoritmo o comportamiento que utilizan. Encapsular el algoritmo en clases Estrategia separadas nos permite variar el algoritmo independientemente de su contexto, haciéndolo más fácil de cambiar, comprender y extender.

3. *Las estrategias eliminan las sentencias condicionales*. El patrón Strategy ofrece una alternativa a las sentencias condicionales para seleccionar el comportamiento deseado. Cuando se juntan muchos comportamientos en una clase es difícil no usar sentencias condicionales para seleccionar el comportamiento correcto. Encapsular el comportamiento en clases Estrategia separadas elimina estas sentencias condicionales.

   Por ejemplo, sin estrategias, el código para dividir un texto en líneas podría parecerse a

```
void Composicion::Reparar () {
   switch (estrategiaDeDivision) {
   case EstrategiaSimple:
      ComponerConComponedorSimple();
      break;
   case EstrategiaTeX:
      ComponerConComponedorTeX();
      break;
   // …
   }
   // si es necesario, combina los
resultados
   // con la composición existente
}
```

   El patrón Estrategia elimina esta sentencia condicional delegando la tarca de división en líneas en el objeto Estrategia:

```
void Composicion::Reparar () {
   _componedor->Componer();
   // si es necesario, combina los
resultados
   // con la composición existente
}
```

   Un código que contiene muchas sentencias condicionales suele indicar la necesidad de aplicar el patrón Estrategia.

4. *Una elección de implementaciones*. Las estrategias pueden proporcionar distintas implementaciones del *mismo* comportamiento. El cliente puede elegir entre estrategias con diferentes soluciones de compromiso entre tiempo y espacio.

5. *Los clientes deben conocer las diferentes Estrategias*, El patrón tiene el inconveniente potencial de que un cliente debe comprender cómo difieren las Estrategias antes de seleccionar la adecuada. Los clientes pueden estar expuestos a cuestiones de implementación. Por tanto, el patrón Strategy debería usarse sólo cuando la variación de comportamiento sea relevante a los clientes.

6. *Costes de comunicación entre Estrategia y Contesto*. La interfaz de Estrategia es compartida por todas las clases EstrategiaConcreta, ya sea el algoritmo que implementa trivial o complejo. Por tanto, es probable que algunos objetos EstrategiaConcreta no usen toda la información que reciben a través de dicha interfaz; las estrategias concretas simples pueden incluso no utilizar nada de dicha información. Eso significa que habrá veces en las que el contexto crea o inicializa parámetros que nunca se usan. Si esto puede ser un problema, necesitaremos un acoplamiento más fuerte entre Estrategia y Contexto.

7. *Mayor número de objetos*. Las estrategias aumentan el número de objetos de una aplicación. A veces se puede reducir este costo implementando las estrategias como objetos sin estado que puedan ser compartidos por el contexto. El contexto mantiene cualquier estado residual, pasándolo en cada petición al objeto Estrategia. Las estrategias compartidas no deberían mantener el estado entre invocaciones. El patrón Flyweight (179) describe este enfoque en más detalle.

IMPLEMENTACIÓN

Examinemos las siguientes cuestiones de implementación:

1. *Definir las interfaces Estrategia y Contesto*. Las interfaces Estrategia y Contexto deben permitir a una EstrategiaConcreta acceder de manera enciente a cualquier dato que ésta necesite del contexto, y viceversa.

   Un enfoque es hacer que Contexto pase los datos como parámetros a las operaciones de Estrategia —en otras palabras, lleva los datos a la estrategia—. Esto mantiene a Estrategia y Contexto desacoplados. Por otro lado. Contexto podría pasar datos a la Estrategia que ésta no necesita.

   Otra técnica consiste en que un contexto se pase a *sí mismo* como argumento, y que la estrategia pida los datos explícitamente al contexto. Como alternativa, la estrategia puede guardar una referencia a su contexto, eliminando así la necesidad de pasar nada. De cualquiera de las dos formas, la estrategia puede pedir exactamente lo que necesita. Pero ahora Contexto debe definir una interfaz más elaborada para sus datos, lo que acopla más estrechamente a Estrategia y Contexto.

   Las necesidades del algoritmo concreto y sus requisitos de dalos determinarán cuál es la mejor técnica.

2. *Estrategias como parámetros de plantillas*. En C++, pueden usarse las plantillas para configurar una clase con una estrategia. Esta técnica sólo se puede aplicar si (1) se puede seleccionar la Estrategia en tiempo de compilación, y (2) no hay que cambiarla en tiempo de ejecución. En este caso, la clase a configurar (por ejemplo, Contexto) se define en una clase plantilla que tiene como parámetro una clase Estrategia:

```
template <class UnaEstrategia>
class Contexto {
   void         Operacion()         {
laEstrategia.HacerAlgoritmo(); }
   // …
private:
```

```
   UnaEstrategia laEstrategia;
};
```

La clase se configura con una clase Estrategia en el momento en que se crea una instancia:

```
class MiEstrategia {
public:
   void HacerAlgoritmo();
};
Contexto<MiEstrategia> unContexto;
```

Con plantillas, no hay necesidad de definir una clase abstracta que defina la interfaz de la Estrategia. Usar Estrategia como un parámetro de plantilla también nos permite enlazar estáticamente una Estrategia a Su Contexto, lo que puede aumentar la eficiencia.

3. *Hacer opcionales los objetos Estrategia*. La clase Contexto puede simplificarse en caso de que tenga sentido *no* tener un objeto Estrategia. Contexto comprueba si tiene un objeto Estrategia antes de acceder a él. En caso de que exista, lo usa normalmente. Si no hay una estrategia, Contexto realiza el comportamiento predeterminado. La ventaja de este enfoque es que los clientes no tienen que tratar con los objetos Estrategia *a menos* que no les sirva el comportamiento predeterminado.

CÓDIGO DE EJEMPLO

A continuación se muestra el código de alto nivel del ejemplo de la sección de la Motivación, que está basado en la implementación de las clases Composición y Componedor de Interviews [LCI+92].

La clase Composición tiene una colección de instancias de Componente, que representan los elementos gráficos y de texto de un documento. Una composición distribuye los objetos componente en líneas usando una instancia de una subclase de Componedor, la cual encapsula una estrategia de división en líneas. Cada

componente tiene un tamaño natural asociado, una dimensión máxima y otra mínima. Estas definen cuánto puede crecer el componente por encima de su tamaño natural y cuánto puede encogerse, respectivamente. La composición pasa estos valores a un componedor, que los usa para determinar la mejor posición para los saltos de línea.

```
class Composicion {
public:
   Composicion(Componedor*);
   void Reparar();
private:
   Componedor* _componedor;
   Componente* _componentes; // la lista de
componentes
   int _contadorComponentes; // el número de
componentes
   int _anchoLinea; // el ancho de linea de
la composición
   int* _saltosLinea;// la posición de los
saltos de
   // línea en los componentes
   int contadorLineas; // el número de líneas
}
```

Cuando se necesita una nueva distribución, la composición le pide a su componedor que determine dónde situar los saltos de línea. La composición pasa al componedor tres arrays que definen el tamaño natural y las dimensiones máximas y mínimas de los componentes. También pasa el número de componentes, el ancho de la línea y un array que rellena el componedor con la posición de cada salto de línea. El componedor devuelve el número de saltos calculados.

La interfaz del Componedor permite que la composición pase a éste toda la información que necesita. Esto es un ejemplo de “llevar los datos a la estrategia”:

```
class Componedor {
public:
   virtual int Componer(
      Coord natural[], Coord estirado[], Coord
encogido[],
```

```
     int contadorComponentes, int anchoLinea,
int saltos!]
  ) = 0;
protected:
  Componedor();
};
```

Nótese que Componedor es una clase abstracta. Las subclases concretas definen estrategias concretas de división en líneas.

La composición llama a su componedor en su operación Reparar. En primer lugar, Reparar inicializa los arrays con el tamaño natural y las dimensiones máxima y mínima de cada componente (omitiremos los detalles de cómo se hace esto en aras de la brevedad). A continuación, llama al componedor para obtener los saltos de línea. Finalmente, distribuye los componentes en función de los saltos de línea (también omitido):

```
void Composicion::Reparar () {
  Coord* natural;
  Coord* maxima;
  Coord* minima;
  int contadorComponentes;
  int* saltos;
  // prepara los arrays con los tamaños
deseados de los componentes
  // …

  // determina dónde van los saltos:
  int contadorSaltos;
  contadorSaltos = _componedor->Componer(
    natural, maxima, minima,
    contadorComponentes, _anchoLinea, saltos
  );

  // coloca los componentes en función de
los saltos
  // …
}
```

Veamos ahora las subclases de Componedor. ComponedorSimple examina los componentes de una línea cada vez, para determinar dónde deberían ir los saltos:

```
class ComponedorSimple : public Componedor {
public:
   ComponedorSimple();

   virtual int Componer(
      Coord natural[], Coord maxima[], Coord
minima[],
      int contadorComponentes, int anchoLinea,
int saltos[]
   );
   // …
};
```

ComponedorTeX usa una estrategia más global. Examina un *párrafo* cada vez, teniendo en cuenta el tamaño y la dimensión máxima de los componentes. También trata de asignar un “color” uniforme al párrafo minimizando el espaciado entre componentes.

```
class ComponedorTeX : public Componedor {
public:
   ComponedorTeX();

   virtual int Componer(
      Coord natural[], Coord maxima[], Coord
minima[],
      int contadorComponentes, int anchoLinea,
int saltos[]
   );
   // …
};
```

ComponedorMatriz separa los componentes en líneas a intervalos regulares.

```
class ComponedorMatriz : public Componedor {
public:
   ComponedorMatriz(int intervalo);

   virtual int Componer(
      Coord natural[], Coord maxima[], Coord
minima[],
      int contadorComponentes, int anchoLinea,
```

```
int saltos[]
   );
   // …
};
```

Estas clases no usan toda la información que se le pasa a Componer. ComponedorSimple no hace uso de la dimensión máxima de los componentes, y sólo tiene en cuenta el ancho natural de éstos. ComponedorTeX usa toda la información que recibe, mientras que ComponedorMatriz no usa nada.

Para crear una instancia de Composición es necesario pasarle el componedor que queremos que use:

```
Composicion* rapida = new Composicion(new
ComponedorSimple);
Composicion* elegante = new Composicion(new
ComponedorTeX);
Composicion* iconos = new Composicion(new
ComponedorMatriz(100));
```

La interfaz de Componedor está cuidadosamente diseñada para permitir toda clase de algoritmos de composición que pudieran implementar las subclases. No queremos tener que cambiar esta interfaz con cada nueva subclase, ya que eso requeriría cambiar las subclases existentes. En general, las interfaces Estrategia y Contexto determinan en qué medida consigue el patrón su propósito.

USOS CONOCIDOS

Tanto ET++ [WGMSS] como Interviews usan estrategias para encapsular diferentes algoritmos de división en líneas tal y como se ha descrito aquí.

En el Sistema RTL System para optimización de código en los compiladores [JML92], las estrategias definen diferentes esquemas de asignación de registros (RegisterAllocator) y de políticas de planificación de juegos de instrucciones (RISCscheduler, CISCscheduler). Esto proporciona flexibilidad a la hora de usar el optimizador para diferentes arquitecturas de máquinas.

El framework de motores de cálculo ET++SwapsManagcr,

calcula los precios de diversos instrumentos financieros [EG92]. Sus abstracciones clave son Instrument y YieldCurve. Los diferentes instrumentos se implementan como subclases de Instrument. YieldCurve calcula los tipos de descuento, que determinan el valor actual de los flujos de caja futuros. Ambas clases delegan parte de su comportamiento en objetos Estrategia. El framework proporciona una familia de clases EstrategiaConcreta para generar flujos de caja, valorar permutas financieras y calcular tipos de descuento. Se pueden crear nuevos motores de cálculo configurando Instrument y YieldCurve con los distintos objetos EstrategiaConcreta. Este enfoque permite combinar y usar las implementaciones existentes de Estrategia así como definir otras nuevas.

Los componentes de Booch [U V90] usan estrategias como argumentos de plantillas. Las clases de colecciones de Booch permiten tres tipos de estrategias de asignación de memoria: gestionada (asignación mediante un pool), controlada (las asignaciones y liberaciones están protegidas por bloqueos) y sin gestionar (el asignador de memoria predeterminado). Estas estrategias se pasan como argumentos de plantilla a una clase de colección cada vez que se crea una instancia de ésta. Por ejemplo, una colección de tamaño variable que usa la estrategia de no gestionar se crea como una UnboundectCollection.

RApp es un sistema para el diseño de circuitos integrados [GA89, AG90]. RApp debe dibujar cables que conectan los distintos subsistemas de un circuito. Los algoritmos que determinan por dónde deben ir dichos cables se definen en RApp como subclases de una clase abstracta Router. Router es una clase Estrategia.

ObjectWindows de Borland [Bor94] usa estrategias en los cuadros de diálogo para asegurar que el usuario introduzca datos válidos. Por ejemplo, los números podrían tener que estar dentro de un intervalo determinado, y un campo numérico sólo debería aceptar dígitos. Validar que una cadena es correcta puede necesitar una tabla de búsqueda.

ObjectWindows usa objetos Validator para encapsular estrategias de validación. Los validadores son ejemplos de objetos Estrategia. Los campos de entrada de datos delegan la validación a un objeto Validator opcional. El cliente asigna un validador a un campo que necesita ser validado (esto constituye un ejemplo de una

estrategia opcional). Cuando se cierra el diálogo, los campos de entrada le piden a sus validadores que validen los datos. La biblioteca de clases proporciona validadores para los casos más comunes, como un Range Validator (validador de intervalo) para números. Se pueden definir fácilmente nuevas estrategias específicas del cliente heredando de la clase Validator.

PATRONES RELACIONADOS

Flyweight (179): los objetos Estrategia suelen ser buenos pesos ligeros.

TEMPLATE METHOD

(Método Plantilla)

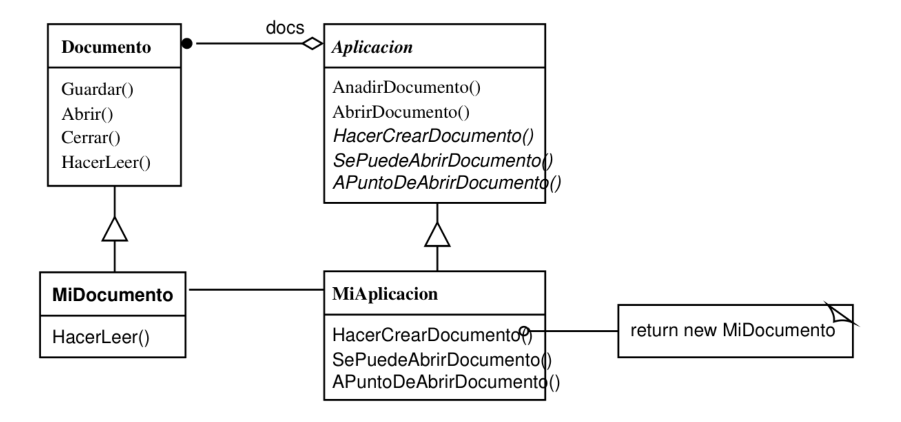
PROPÓSITO

Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.

MOTIVACIÓN

Pensemos en un framework que proporciona las clases Aplicación y Documento. La clase Aplicación es la responsable de abrir el documento guardado en un formato externo, como por ejemplo un fichero. Un objeto Documento representa la información de un documento una vez que éste ha sido leído del fichero.

Las aplicaciones construidas con el framework pueden heredar de Aplicación o de Documento para adaptarse a necesidades específicas. Por ejemplo, una aplicación de dibujo define las subclases AplicacionDeDibujo y DocumentoDeDibujo; una aplicación de hoja de cálculo define las subclases AplicacionDeHojaDeCalculo y DocumentoDeHojaDeCalculo.



La clase Aplicación define el algoritmo para abrir y leer un documento en su operación AbrirDocumento:

```
void Aplicacion::AbrirDocumento (const char*
nombre) {
   if (!SePuedeAbrirDocumento(nombre)) {
      // no se puede abrir este documento
      return;
   }
   Documento* doc = HacerCrearDocumento();
   if (doc) {
      _docs->AnadirDocumento(doc);
      APuntoDeAbrirDocumento(doc);
      doc->Abrir();
      doc->HacerLeer();
   }
}
```

AbrirDocumento define cada paso para abrir el documento. Comprueba si el documento puede abrirse, crea el objeto Documento específico de la aplicación, lo añade a su conjunto de documentos y lee el Documento de un fichero.

Llamaremos a AbrirDocumento un método plantilla. Un método plantilla define un algoritmo en términos de operaciones abstractas que las subclases deben redefinir para proporcionar un determinado comportamiento. Las subclases do Aplicación definen los pasos del algoritmo que comprueban si el documento puede abrirse (SePuedeAbrirDocumento) y que crean el Documento

(HacerCrearDocumento). Las clases Documento definen el paso que lee el documento (HacerLeer). El método plantilla también define una operación que permite que las subclases de Aplicación sepan cuándo se va abrir el documento (APuntoDeAbrirDocumento).

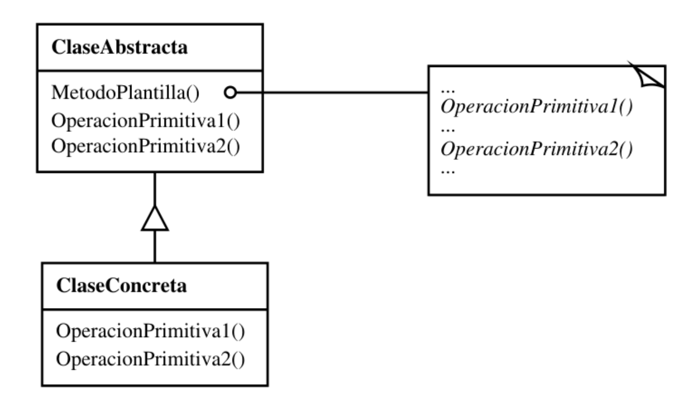
Al definir algunos de los pasos de un algoritmo usando operaciones abstractas, el método plantilla fija su ordenación, pero permite que las subclases de Aplicación y Documento modifiquen dichos pasos para adecuarse a sus necesidades.

APLICABILIDAD

El patrón Template Method debería usarse

- para implementar las partes de un algoritmo que no cambian y dejar que sean las subclases quienes implementen el comportamiento que puede variar.
- cuando el comportamiento repetido de varias subclases debería factorizarse y ser localizado en una clase común para evitar el código duplicado. Ésta es una buena idea de "refactorizar para generalizar", tal como la describen Opdyke y Johnson [OJ93]. En primer lugar identificamos las diferencias en el código existente y a continuación separamos dichas diferencias en nuevas operaciones. Por último, sustituimos el código que cambia por un método que llama a una de estas nuevas operaciones.
- para controlar las extensiones de las subclases. Podemos definir un método plantilla que llame a operaciones "de enganche " (véanse las Consecuencias) en determinados puntos, permitiendo así las extensiones sólo en esos puntos.

ESTRUCTURA



PARTICIPANTES

- **ClaseAbstracta** (Aplicación)
  - define operaciones primitivas abstractas que son definidas por las subclases para implementar los pasos de un algoritmo.
  - implementa un método plantilla que define el esqueleto de un algoritmo. El método plantilla llama a las operaciones primitivas así como a operaciones definidas en ClaseAbstracta o a las de otros objetos.
- **ClaseConcreta** (MiAplicacion)
  - Implementa las operaciones primitivas para realizar los pasos del algoritmo específicos de las subclases.

COLABORACIONES

ClaseConcreta se basa en ClaseAbstracta para implementar los pasos de un algoritmo que no cambian.

CONSECUENCIAS

Los métodos plantilla son una técnica fundamental de reutilización de código. Son particularmente importantes en las bibliotecas de clases, ya que son el modo de factorizar y extraer el comportamiento común de las clases de la biblioteca.

Los métodos plantilla llevan a una estructura de control

invertido que a veces se denomina “el principio de Hollywood”, es decir, “No nos llame, nosotros le llamaremos” (Swe85). Esto se refiere a cómo una clase padre llama a las operaciones de una subclase y no al revés.

Los métodos plantilla llaman a los siguientes tipos de operaciones:

- operaciones concretas (ya sea de la ClaseConcreta o de las clases cliente);
- operaciones concretas de ClaseAbstracta (es decir, operaciones que suelen ser útiles para las subclases);
- operaciones primitivas (es decir, operaciones abstractas);
- métodos de fabricación (véase el patrón Factory Method (99)); y
- operaciones de enganche, que proporcionan el comportamiento predeterminado que puede ser modificado por las subclases si es necesario. Una operación de enganche normalmente no hace nada por omisión.

Es importante que los métodos plantilla especifiquen qué operaciones son enganches (que *pueden* ser redefinidas) y cuáles son operaciones abstractas (que *deben* ser redefinidas). Para reutilizar una clase abstracta apropiadamente, los escritores de las subclases deben saber qué operaciones están diseñadas para ser redefinidas.

Una subclase puede *extender* el comportamiento de una operación de una clase padre redefiniendo la operación y llamando explícitamente a la operación del padre:

```
void ClaseDerivada::Operacion () {
  //      ClaseDerivada      extiende      el
comportamiento
  ClasePadre::Operacion();
}
```

Desgraciadamente, es fácil olvidarse de llamar a la operación heredada. Podemos transformar esta operación en un método plantilla que le dé control al padre sobre cómo éste puede ser

extendido por las subclases. La idea es llamar a una operación de enganche desde un método plantilla en la clase padre. Las subclases pueden entonces redefinir esta operación de enganche:

```
void ClasePadre::Operacion () {
  // comportamiento de ClasePadre
  OperacionDeEnganche();
}
```

OperacionDeEnganche no hace nada en la ClasePadre:

```
void ClasePadre::OperacionDeEnganche () { }
```

Las subclases redefinen OperacionDeEnganche para extender su comportamiento:

```
void ClaseDerivada::OperacionDeEnganche () {
  // extensión de la clase derivada
}
```

IMPLEMENTACIÓN

Merece la pena tener en cuenta tres detalles de implementación:

1. *Usar el control de acceso de* C++. En C++, las operaciones primitivas a las que llama un método plantilla pueden ser declaradas como miembros protegidos. Esto garantiza que sólo pueden ser llamadas por el método plantilla. Las operaciones primitivas que *deben* ser redefinidas se declaran como virtuales puras. El método plantilla en sí no debería ser redefinido; por tanto podemos hacer que sea una función miembro no virtual.
2. *Minimizar las operaciones primitivas*. Un objetivo importante para diseñar métodos plantilla es minimizar el número de operaciones primitivas que una subclase debe redefinir para dar cuerpo al algoritmo. Cuantas más operaciones necesiten

ser redefinidas, más tediosas se vuelven las cosas para los clientes.

3. *Convenios de nominación*. Se pueden identificar las operaciones que deberían ser redefinidas añadiendo un prefijo a sus nombres. Así, por ejemplo, el framework MacApp para aplicaciones de MacIntosh [App89] añade a los nombres de los métodos plantilla el prefijo “Do-”: “DoCreateDocument”, “DoRead”, etcétera.

CÓDIGO DE EJEMPLO

El siguiente código en C++ muestra cómo puede una clase padre obligar a sus subclases a respetar un invariante. El ejemplo procede del AppKit de NeXT AppKit [Add94]. Pensemos en una clase Vista que permite dibujar en la pantalla. Vista hace cumplir el invariante de que sus subclases pueden dibujar en una vista sólo después de que ésta ha recibido el “foco”, lo que requiere establecer de forma apropiada parte del estado del dibujo (como, por ejemplo, las fuentes y los colores).

Podemos usar un método plantilla, Mostrar, para establecer dicho estado. Vista define dos operaciones concretas, AsignarFoco y QuitarFoco, que establecen y limpian el estado del dibujo, respectivamente. La operación HacerMostrar de Vista es quien realiza el dibujado real. Mostrar llama a AsignarFoco antes de HacerMostrar para establecer el estado del dibujo; después, Mostrar llama a QuitarFoco para liberar dicho estado.

```
void Vista::Mostrar () {
  AsignarFoco();
  HacerMostrar();
  QuitarFoco();
}
```

Para conservar el invariante, los clientes de Vista siempre llaman a Mostrar, y las subclases de Vista siempre redefinen HacerMostrar.

HacerMostrar no hace nada en Vista:

```
void Vista::HacerMostrar () { }
```

Las subclases lo redefinen para añadir su comportamiento de dibujado específico:

```
void MiVista::HacerMostrar () {
   // muestra los contenidos de la vista
}
```

USOS CONOCIDOS

Los métodos plantilla son tan fundamentales que pueden encontrarse en casi cualquier clase abstracta. Wirfs-Brock *et al.* [WBWW90, WBJ90] proporcionan una buena discusión de los métodos plantilla.

PATRONES RELACIONADOS

Los Métodos de Fabricación (99) se llaman muchas veces desde métodos plantilla. En el ejemplo de la sección de Motivación, el método de fabricación HacerCrearDocumento es llamado por el método plantilla AbrirDocumento.

Strategy (289): los métodos plantilla usan la herencia para modificar una parte de un algoritmo. Las estrategias usan delegación para variar el algoritmo completo.

VISITOR (Visitante)

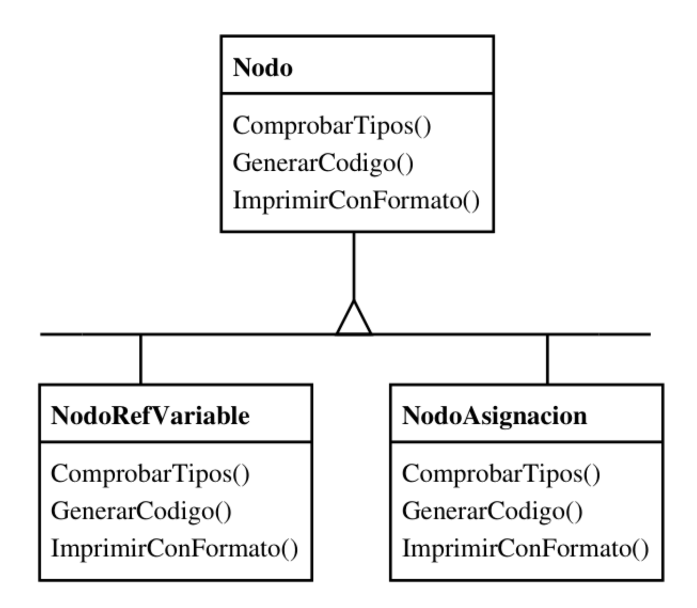
PROPÓSITO

Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

MOTIVACIÓN

Pensemos en un compilador que representa programas como árboles sintácticos abstractos. Necesitaremos realizar operaciones sobre dichos árboles sintácticos abstractos para llevar a cabo el análisis “semántico estático", como comprobar que todas las variables están definidas. También necesitaremos generar código. Por tanto, podríamos definir operaciones para la comprobación de tipos, la optimización de código, el análisis de flujo, comprobar que se asignan valores a las variables antes de su uso, etcétera. Más aún, podríamos usar los árboles sintácticos abstractos para imprimir con formato, reestructurar el programa, instrumentación de código o calcular diferentes métricas de un programa.

La mayoría de estas operaciones necesitarán tratar a los nodos que representan sentencias de asignación de forma distinta que a les que representan variables o expresiones aritméticas. Por tanto, habrá una clase para sentencias de asignación, otra para los accesos a variables, otra para expresiones aritméticas y así sucesivamente. El conjunto de clases de nodos depende del lenguaje que está siendo compilado, por supuesto, pero no cambia mucho para un lenguaje dado.



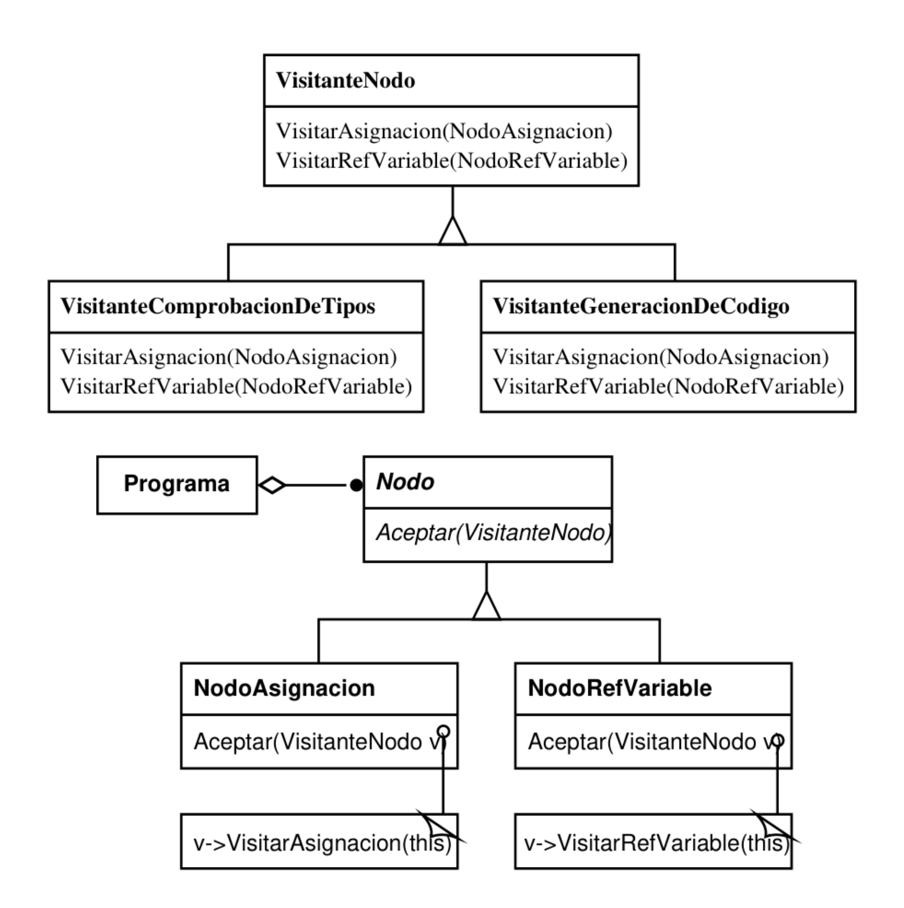
Este diagrama muestra parte de la jerarquía de clases de Nodo. El problema aquí es que distribuir todas estas operaciones a través de las distintas clases de nodos conduce a un sistema que es difícil de comprender, mantener y cambiar. Será confuso tener la comprobación de tipos mezclada con el código de impresión o con el de análisis de flujo. Además, añadir una nueva operación normalmente obliga a recompilar todas estas clases. Sería mejor si cada nueva operación pudiera ser añadida por separado y las clases de nodos fuesen independientes de las operaciones que se aplican sobre ellas.

Podemos lograr ambas cosas empaquetando las operaciones relacionadas de las distintas clases en un objeto aparte, llamado visitante, al que se le pasen los elementos del árbol sintáctico abstracto a medida que va siendo recorrido. Cuando un elemento "acepta" al visitante, le envía una petición que codifica la clase del elemento. También incluye al elemento como argumento. El visitante ejecutará entonces la operación para ese elemento —la operación que solía estar en la clase del elemento—.

Por ejemplo, un compilador que no usara visitantes podría comprobar los tipos de un procedimiento llamando a la operación ComprobarTipos sobre su árbol sintáctico abstracto. Cada unos de los nodos implementaría ComprobarTipos llamando a

ComprobarTipos sobre sus componentes (*véase* el diagrama de clases precedente). Si la comprobación de tipos de un procedimiento usara visitantes, entonces crearía un objeto VisitanteComprobacionDeTipos y llamaría a la operación Aceptar sobre el árbol sintáctico abstracto con ese objeto como argumento. Cada uno de los nodos implementaría Aceptar llamando a su vez al visitante: un nodo de asignación llama a la operación VisitarAsignacion del visitante, mientras que una referencia a una variable llama a VisitarReferenciaAVariable. Lo que antes era la operación ComprobarTipos de la clase NodoAsignacion ahora es la operación VisitarAsignacion de VisitanteComprobacionDeTipos.

Para hacer que los visitantes sirvan para algo más que simplemente la comprobación de tipos necesitamos una clase padre abstracta, VisitanteNodo, para todos los visitantes de un árbol sintáctico abstracto. VisitanteNodo debe declarar una operación para clase de nodo. Una aplicación que necesite calcular métricas de programas definirá nuevas subclases de VisitanteNodo y ya no necesitará añadir código específico de una aplicación a las clases de los nodos. El patrón Visitor encapsula las operaciones de cada fase de compilación en un Visitante asociado a esa fase.



Con el patrón Visitor, definimos dos jerarquías de clases: una para los elementos sobre los que se opera (la jerarquía de Nodo) y otra para los visitantes que definen operaciones sobre los elementos (la jerarquía de VisitanteNodo). La forma de crear una nueva operación es añadiendo una nueva subclase a la jerarquía de clases de los visitantes. Siempre y cuando no cambie la gramática aceptada por el compilador (es decir, siempre que no tengamos que añadir nuevas subclases de Nodo), podemos añadir nueva funcionalidad simplemente definiendo nuevas subclases de VisitanteNodo.

APLICABILIDAD

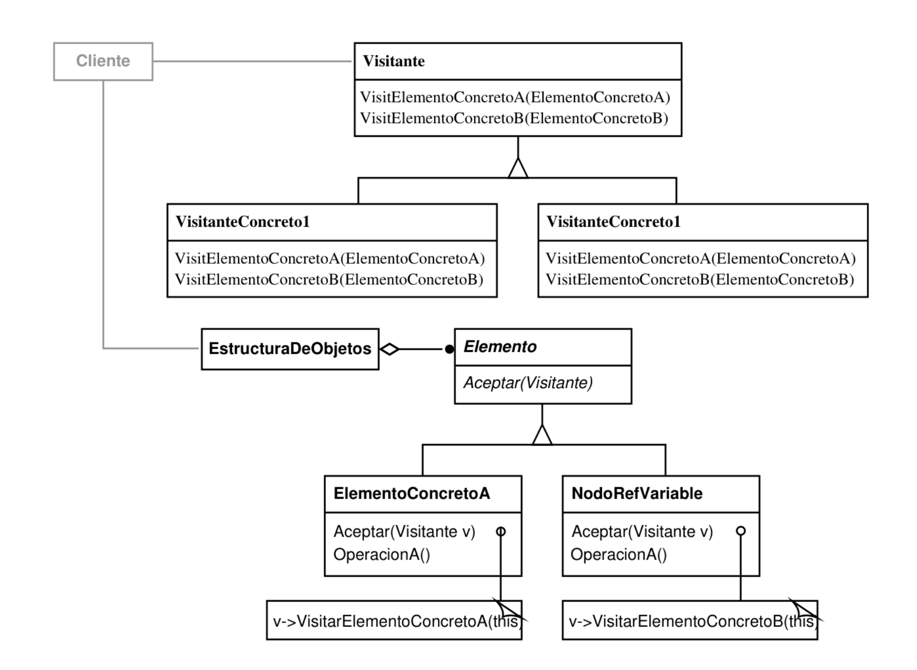
Úsese el patrón Visitor cuando

- una estructura de objetos contiene muchas clases de objetos

con diferentes interfaces, y queremos realizar operaciones sobre esos elementos que dependen de su clase concreta.

- se necesitan realizar muchas operaciones distintas y no relacionadas sobre objetos de una estructura de objetos, y queremos evitar "contaminar" sus clases con dichas operaciones. El patrón Visitor permite mantener juntas operaciones relacionadas definiéndolas en una clase. Cuando la estructura de objetos es compartida por varias aplicaciones, el patrón Visitor permite poner operaciones sólo en aquellas aplicaciones que las necesiten.
- las clases que definen la estructura de objetos rara vez cambian, pero muchas veces queremos definir nuevas operaciones sobre la estructura. Cambiar las clases de la estructura de objetos requiere redefinir la interfaz para todos los visitantes, lo que es potencialmente costoso. Si las clases de la estructura cambian con frecuencia, probablemente sea mejor definir las operaciones en las propias clases.

ESTRUCTURA



PARTICIPANTES

- **Visitante** (VisitanteNodo)
  - declara una operación Visitar para cada clase de operación ElementoConcreto de la estructura de objetos. El nombre y signatura de la operación identifican a la clase que envía la petición Visitar al visitante. Eso permite al visitante determinar la clase concreta de elemento que está siendo visitada. A continuación el visitante puede acceder al elemento directamente a través de su interfaz particular.
- **VisitanteConcreto** (VisitanteComprobacionDeTipos)
  - implementa cada operación declarada por Visitante. Cada operación implementa un fragmento del algoritmo definido para la clase correspondiente de la estructura. VisitanteConcreto proporciona el contexto para el algoritmo y guarda su estado local. Muchas veces este estado acumula resultados durante el recorrido de la estructura.
- **Elemento** (Nodo)
  - define una operación Aceptar que toma un visitante como argumento.
- **ElementoConcreto** (NodoAsignacion, NodoRefVariable)
  - implementa una operación Aceptar que toma un visitante como argumento.
- **EstructuraDeObjetos** (Programa)
  - puede enumerar sus elementos.
  - puede proporcionar una interfaz de alto nivel para permitir al visitante visitar a sus elementos.
  - puede ser un compuesto (*véase* el patrón Composite (151)) o una colección, como una lista o un conjunto.

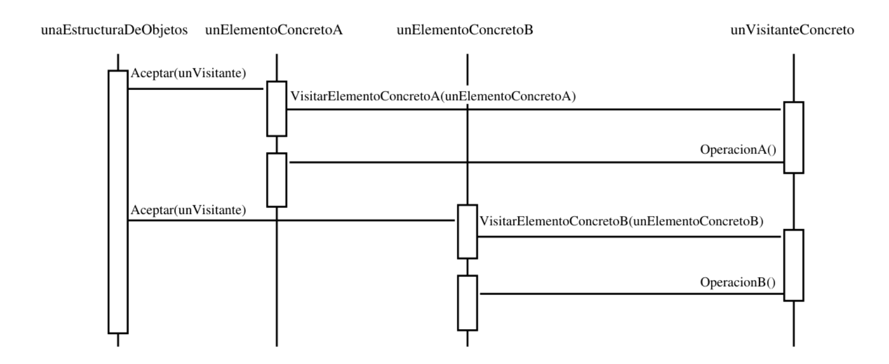
COLABORACIONES

- Un cliente que usa el patrón Visitor debe crear un objeto

VisitanteConcreto y a continuación recorrer la estructura, visitando cada objeto con el visitante.

- Cada vez que se visita a un elemento, éste llama a la operación del Visitante que se corresponde con su clase. El elemento se pasa a sí mismo como argumento de la operación para permitir al visitante acceder a su estado, en caso de que sea necesario.

El siguiente diagrama de interacción ilustra las colaboraciones entre una estructura de objetos, un visitante y dos elementos:



CONSECUENCIAS

Algunas de las ventajas c inconvenientes del patrón Visitor son las siguientes:

1. *El visitante facilita añadir nuevas operaciones*. Los visitantes facilitan añadir nuevas operaciones que dependen de los componentes de objetos complejos. Podemos definir una nueva operación sobre una estructura simplemente añadiendo un nuevo visitante. Si, por el contrario, extendiésemos la funcionalidad sobre muchas clases, habría que cambiar cada clase para definir una nueva operación.

2. *Un visitante agrupa operaciones relacionadas y separa las que no lo están*. El comportamiento similar no está desperdigado por las clases que definen la estructura de

objetos; esté localizado en un visitante. Las partes de comportamiento no relacionadas se dividen en sus propias subclases del visitante. Esto simplifica tanto las clases que definen los elementos como los algoritmos definidos por los visitantes. Cualquier estructura de datos específica de un algoritmo puede estar oculta en el visitante.

3. *Es difícil añadir nuevas clases de ElementoConcreto*. El patrón Visitor hace que sea complicado añadir nuevas subclases de Elemento. Cada ElementoConcreto nuevo da lugar a una nueva operación abstracta del Visitante y a su correspondiente implementación en cada clase VisitanteConcreto. A veces se puede proporcionar en Visitante una implementación predeterminada que puede ser heredada por la mayoría de los visitantes concretos, pero esto representa una excepción más que una regla.

   Por tanto la cuestión fundamental a considerar a la hora de aplicar el patrón Visitor es si es más probable que cambie el algoritmo aplicado sobre una estructura de objetos o las clases de los objetos que componen la estructura. La jerarquía de clases de Visitante puede ser difícil de mantener cuando se añaden nuevas clases de ElementoConcreto con frecuencia. En tales casos, es probablemente más fácil definir las operaciones en las clases que componen la estructura. Si la jerarquía de clases de Elemento es estable pero estamos continuamente añadiendo operaciones o cambiando algoritmos, el patrón Visitor nos ayudaría a controlar dichos cambios.

4. *Visitar varias jerarquías de clases*. Un iterador (*véase* el patrón Iterator(237)) puede visitar a los objetos de una estructura llamando a sus operaciones a medida que los recorre. Pero un iterador no puede trabajar en varias estructuras de objetos con distintos tipos de elementos. Por ejemplo, la interfaz Iterador definida en la página 242 puede acceder únicamente a objetos del tipo Elemento:

```
template «class Elementos
```

```
class Iterador {
   // …
   Elemento ElementoActual() const;
};
```

Esto implica que todos los elementos que el iterador puede visitar tienen una clase padre común Elemento.

El patrón Visitor no tiene esta restricción. Puede visitar objetos que no tienen una clase padre común. Se puede añadir cualquier tipo de objeto a la interfaz de un Visitante, Por ejemplo, en

```
class Visitante {
public:
     // …
     void VisitarMiTipo(MiTipo*);
     void VisitarTuTipo(TuTipo*);
};
```

MiTipo y TuTipo no tienen por qué estar relacionados en modo alguno a través de la herencia.

5. *Acumular el estado*. Los visitantes pueden acumular estado a medida que van visitando cada elemento de la estructura de objetos. Sin un visitante, este estado se pasaría como argumentos extra a las operaciones que realizan el recorrido, o quizá como variables globales.

6. *Romper la encapsulación*. El enfoque del patrón Visitor asume que la interfaz de ElementoConcreto es lo bastante potente como para que los visitantes hagan su trabajo. Como resultado, el patrón suele obligamos a proporcionar operaciones públicas que accedan al estado interno de un elemento, lo que puede comprometer su encapsulación.

IMPLEMENTACIÓN

Cada estructura de objetos tendrá una clase Visitante asociada. Esta clase visitante abstracta declara una operación

VisitarElementoConcreto para cada clase de ElementoConcreto que define la estructura de objetos. Cada operación Visitar del Visitante declara como argumento un ElementoConcreto en particular, permitiendo al Visitante acceder directamente a la interfaz del ElementoConcreto. Las clases VisitanteConcreto redefinen cada operación Visitar para implementar el comportamiento específico del visitante para la correspondiente clase ElementoConcreto.

```
La clase Visitante se declararía así en C++:
class Visitante {
public:
   virtual void VisitarElementoA(ElementoA*);
   virtual void VisitarElementoB(ElementoB*);

   // y así para otros elementos concretos
protected:
   Visitante();
};
```

Cada clase de ElementoConcreto implementa una operación Aceptar que llama a la operación Visitar… del visitante correspondiente a ese ElementoConcreto. De modo que la operación que es llamada al final depende tanto de la clase del elemento como de la clase del visitante **[62]** .

Los elementos concretos se declaran como

```
class Elemento {
public:
   virtual ~Elemento();
   virtual void Aceptar(Visitante&) = 0;
protected:
   Elemento();
};

class ElementoA : public Elemento {
public:
   ElementoA();
   virtual   void   Aceptar(Visitante&   v)   {
v.VisitarElementoA(this); }
};

class ElementoB : public Elemento {
```

```
public:
   ElementoB();
   virtual   void   Aceptar(Visitante&   v)   {
v.VisitarElementoB(this); }
};
```

Una clase ElementoCompuesto podría implementar Aceptar como sigue:

```
class ElementoCompuesto : public Elemento {
public:
   virtual void Aceptar(Visitante&);
private:
   Lista<Elemento*>* _hijos;
};

void  ElementoCompuesto::Aceptar  (Visitante&
v) {
   IteradorLista<Elemento*> i(_hijos);
   for     (i.Primero();     !i.HaTerminado();
i.Siguiente()) {
      i.ElementoActual()->Aceptar(v);
   }
   v.VisitarElementoCompuesto(this);
}
```

Éstas son otras dos cuestiones de implementación que surgen al aplicar el patrón Visitor:

1. *Doble despacho*. El patrón Visitor nos permite añadir operaciones a clases sin modificar éstas. Esto se logra mediante una técnica llamada **doble-despacho**, la cual es muy conocida. De hecho, algunos lenguajes de programación la permiten directamente (por ejemplo, CLOS). Otros lenguajes, como C++ y Smalltalk, permiten el **despacho-único**.

   En lenguajes de despacho-único, dos son los criterios que determinan qué operación satisfará una petición: el nombre de la petición y el tipo del receptor. Por ejemplo, la operación a la que llamará una petición a

GenerarCodigo dependerá del tipo de objeto nodo al que se le pida. En C++, llamar a GenerarCodigo sobre una instancia de NodoRefVariable llamará a NodoRefVariable::GenerarCodigo (que genera código para una referencia a una variable). Llamar a GenerarCodigo sobre un NodoAsignacion llamará a NodoAsignacion::GenerarCodigo (que generará código para una asignación). La operación que se ejecuta depende tanto del tipo del solicitante como del tipo del receptor.

"Doble-despacho" simplemente significa que la operación que se ejecuta depende del tipo del solicitante y de los tipos de *dos* receptores. Aceptar es una operación de doble-despacho. Su significado depende de dos tipos: el del Visitante y el del Elemento. El doble-despacho permite a los visitantes solicitar diferentes operaciones en cada clase de elemento[63]. Ésta es la clave del patrón Visitor: la operación que se ejecuta depende tanto del tipo del Visitante como del tipo del Elemento visitado. En vez de enlazar las operaciones estáticamente en la interfaz de Elemento, podemos fusionar las operaciones en un Visitante y usar Aceptar para hacer el enlace en tiempo de ejecución. Extender la interfaz de Elemento consiste en definir una nueva subclase de Visitante en vez de muchas nuevas subclases de Elemento.

2. *¿Quién es el responsable de recorrer la estructura de objetos*? Un visitante debe visitar cada elemento de la estructura de objetos. La cuestión es, ¿cómo lo logra? Podemos poner la responsabilidad del recorrido en cualquiera de estos tres sitios: en la estructura de objetos, en el visitante o en un objeto iterador aparte (*véase* el patrón Iterator (237)).

   Muchas veces es la estructura de objetos la responsable de la iteración. Una colección simplemente iterará sobre sus elementos, llamando a la operación Aceptar de cada uno. Un compuesto generalmente se recorrerá a sí mismo haciendo que cada operación Aceptar recorra los hijos del

elemento y llame a Aceptar sobre cada uno de ellos, recursivamente.

Otra solución es usar un iterador para visitar los elementos. En C++, podríamos usar un iterador interno o externo, dependiendo de qué está disponible y qué es más eficiente. En Smalltalk, normalmente usarnos un iterador interno mediante do: y un bloque. Puesto que los iteradores internos son implementados por la estructura de objetos, usar un iterador interno se parece mucho a hacer que sea la estructura de objetos la responsable de la iteración. La principal diferencia estriba en que un iterador interno no provocará un doble-despacho —llamará a una operación del *visitante* con un *elemento* como argumento, frente a llamar a una operación del *elemento* con el *visitante* como argumento—. Pero resulta sencillo usar el patrón Visitor con un iterador interno si la operación del visitante simplemente llama a la operación del elemento sin recursividad.

Incluso se podría poner el algoritmo de recorrido en el visitante, si bien en ese caso acabaríamos duplicando el código del recorrido en cada VisitanteConcreto de cada agregado ElementoConcreto. La principal razón para poner la estrategia de recorrido en el visitante es implementar un recorrido especialmente complejo, que dependa de los resultados de las operaciones de la estructura de objetos. En el Código de Ejemplo se verá un ejemplo para este caso.

CÓDIGO DE EJEMPLO

Como los visitantes suelen asociarse a compuestos, usaremos la clase Equipo que se definió en el Código de Ejemplo del patrón Composite (151) para ilustrar el patrón Visitor. Usaremos el patrón Visitor para definir operaciones para realizar el inventario de materiales y calcular el coste total de un equipo. Las clases Equipo son tan sencillas que realmente no sería necesario usar el patrón Visitor, pero lo haremos así para mostrar qué implicaciones conlleva la implementación de este patrón.

A continuación se muestra de nuevo la clase Equipo del Composite (151). La hemos aumentado con la operación Aceptar para que pueda funcionar con un visitante.

```
class Equipo {
public:
   virtual ~Equipo();

   const char* Nombre() { return _nombre; }

   virtual Vatio Potencia();
   virtual Moneda PrecioNeto();
   virtual Moneda PrecioConDescuento() ;

   virtual void Aceptar(VisitanteEquipo&);
protected:
   Equipo(const char*);
private:
   const char* _nombre;
};
```

Las operaciones de Equipo devuelven los atributos de un equipo, tales como su consumo de potencia y su coste. Las subclases redefinen estas operaciones de forma apropiada a cada tipo de equipo (por ejemplo, chasis, unidades y placas base).

La clase abstracta para todos los visitantes de equipos tiene una función virtual para cada subclase de equipo, como se muestra a continuación. Todas las funciones virtuales no hacen nada por omisión.

```
class VisitanteEquipo {
public:
   virtual ~VisitantoEquipo();

   virtual                                void
VisitarDisquetera(Disquetera*);
   virtual void VisitarTarjeta(Tarjeta*);
   virtual void VisitarChasis(Chasis*);
   virtual void VisitarBus(Bus*);

   //  y  así  para  el  resto  do  subclases
concretas de Equipo
protected:
```

```
    VisitanteEquipo();
};
```

Las subclases de Equipo definen Aceptar básicamente de la misma forma: llamando a la operación de VisitanteEquipo que se corresponda con la clase que recibe la petición Aceptar, como en:

```
void  Disquetera::Aceptar  (VisitanteEquipo&
visitante) {
    visitor.VisitarDisquetera(this);
}
```

Los equipos que contienen otros equipos (en concreto, las subclases de EquipoCompuesto en el patrón Composite) implementan Aceptar iterando sobre sus hijos y llamando a Aceptar sobre cada uno de ellos. A continuación llama a la operación Visitar como siempre. Por ejemplo, Chasis::Aceptar podría recorrer todas las partes del Chasis como sigue:

```
void    Chasis::Aceptar    (VisitanteEquipo&
visitante) {
    for {
        IteradorLista <Equipo*> i(_ partes);
        !i.HaTerminado();
        i.Siguiente()
    ) {
        i.ElementoActual()->Aceptar(visitar);
    }
    visitante.VisitarChasis(this);
}
```

Las subclases de VisitanteEquipo definen algoritmos concretos sobre la estructura de equipos. El VisitantePrecio calcula el coste de la estructura de equipos, calculando el precio neto de todos los equipos simples (por ejemplo, las disqueteras) y el precio con descuento de todos los equipos compuestos (como los chasis y buses).

```
class      VisitantePrecio      :      public
VisitanteEquipo {
public:
   VisitantePrecio();

   Moneda& ObtenerPrecioTotal();

   virtual                                void
VisitarDisquetera(Disquetera*);
   virtual void VisitarTarjeta(Tarjeta*);
   virtual void VisitarChasis(Chasis*);
   virtual void VisitarBus(Bus*);
   // …
private:
   Moneda _total;
};

void       VisitantePrecio::VisitarDisquetera
(Disquetera* e) {
   _total += e->PrecioNeto();
}

void  VisitantePrecio::VisitarChasis  (Chasis*
e) {
   _total += e->PrecioConDescuento();
}
```

VisitantePrecio calculara el coste total de todos los nodos de la estructura de equipos. Nótese que VisitantePrecio elige la política de precios apropiada para una clase de equipo despachando a la correspondiente función miembro. Y lo que es más, podemos cambiar la política de precios de una estructura de equipo simplemente cambiando la clase VisitantePrecio.

Podemos definir un visitante para realizar un inventario como sigue:

```
class     VisitanteInventario     :     public
VisitanteEquipo {
public:
   VisitanteInventario();

   Inventario& ObtenerInventario() ;

   virtual                                void
VisitarDisquetera(Disquetera*);
```

```
    virtual void VisitarTarjeta(Tarjeta*);
    virtual void VisitarChasis(Chasis*);
    virtual void VisitarBus(Bus*);
    // …

private:
    Inventario _inventario;
};
```

El VisitanteInventario acumula los totales de cada tipo de equipo de la estructura de objetos. VisitanteInventario usa una clase Inventario que define una interfaz para añadir equipamiento (que no nos molestaremos en definir aquí).

```
void   VisitanteInventario::VisitarDisquetera
(Disquetera* e) {
    _inventario.Acumular(e);
}

void       VisitanteInventario::VisitarChasis
(Chasis* e) {
    _inventario.Acumular(e);
}
```

Así es como podemos usar un VisitanteInventario en una estructura de equipos:

```
Equipo* componente;
VisitanteInventario visitante;

componente->Aceptar(visitor);
cout << "Inventario "
     << componente->Nombre()
     << visitante.ObtenerInventario();
```

Ahora veremos cómo implementar el ejemplo de Smalltalk del patrón Interpreter (*véase* la página 229) con el patrón Visitor. Como en el ejemplo anterior, éste es tan pequeño que el Visitante probablemente no nos aporte gran cosa, pero proporciona un buen ejemplo de cómo usar el patrón. Además, muestra una situación en

la que la iteración es responsabilidad del visitante.

La estructura de objetos (expresiones regulares) se compone de cuatro clases, y todas ellas tienen un método aceptar: que toma un visitante como argumento. En la clase ExpresionSecuencia, el método aceptar: se define como

```
aceptar: unVisitante
     ^ unVisitante visitarSecuencia: self
```

En la clase ExpresionRepetir, el método aceptar: envía el mensaje visitarRepetir:. En la clase ExpresionAlternativa, envía el mensaje visitarAlternativa:. En la clase ExpresionLiteral, envía el mensaje visitarLiteral:.

Las cuatro clases también deben tener funciones de acceso que pueda usar el visitante. Para ExpresionSecuencia éstas son expresionl y expresion2; para ExpresionAlternativa son alternativa1 y alternativa2; para ExpresionRepetir es su repetición; y para ExpresionLiteral sus componentes.

La clase VisitanteConcreto es VisitanteReconocedorER. Es la responsable del recorrido porque su algoritmo de recorrido es irregular. La mayor irregularidad es que una ExpresionRepetir recorrerá repetidamente su componente. La clase VisitanteReconocedorER tiene una variable de instancia estadoEntrada. Sus métodos son, en esencia, los mismos que los métodos reconocer: de las clases de expresiones del patrón Interpreter, salvo que éstas sustituyen el argumento llamado estadoEntrada por el nodo con la expresión que está siendo reconocida. En cualquier caso, siguen devolviendo el conjunto de flujos que puede reconocer la expresión para identificar el estado actual.

```
visitarSecuencia: expSecuencia
   estadoEntrada  :=  expSecuencia  expresion1
aceptar: self.
   ^ expSecuencia expresion2 aceptar: self.

visitarRepetir: expRepetir
   ¦ estadoFinal ¦
```

```
   estadoFinal := estadoEntrada copy.
   [estadoEntrada isEmpty]
      whileFalse:
         [estadoEntrada      :=      expRepetir
repeticion aceptar: self.
         estadoFinal addAll: estadoEntrada].
      ^ estadoFinal

visitarAlternativa: expAlternativa
   ¦ estadoFinal estadoOriginal ¦
   estadoOriginal := estadoEntrada.
   estadoFinal := expAlternativa alternativa1
aceptar: self.
   estadoEntrada := estadoOriginal.
   estadoFinal    addAll:    (expAlternativa
alternativa2 aceptar: self).
   ^ estadoFinal

visitarLiteral: expLiteral
   ¦ estadoFinal tStream ¦
   estadoFinal := Set new.
   estadoEntrada
      do:
         [:stream ¦ tStream := stream copy.
            (tStream nextAvailable:
               expLiteral componentes size
            ) = expLiteral componentes
               ifTrue:    [estadoFinal    add:
tStream]
         ].
      ^ estadoFinal
```

USOS CONOCIDOS

El compilador Smalltalk-80 tiene una clase Visitante llamada ProgramNodeEnumerator. Se usa sobre todo para los algoritmos que analizan el código fuente. No se usa para generación de código o impresión con formato, aunque se podría.

IRIS Inventor [Str93] es un toolkit para desarrollar aplicaciones gráficas en
3-D.
Inventor representa una escena tridimensional como una jerarquía de nodos, cada uno de los cuales representa o bien un objeto geométrico o bien uno de sus atributos. Operaciones como mostrar

una escena o establecer una acción para un evento de entrada necesitan recorrer esta estructura de varias formas. Inventor lleva a cabo esto usando visitantes llamados “acciones”. Hay diferentes visitantes para la visualización, el manejo de eventos, la búsqueda, guardar en ficheros o determinar las cajas limítrofes.

Para facilitar la adición de nuevos nodos, Inventor implementa un esquema de doble-despacho en C++. Este esquema se basa en la información de tipos en tiempo de ejecución y en una tabla bidimensional en la que las filas representan visitantes y las columnas clases de nodos. Las casillas guardan un puntero a la función asignada a esas clases de visitante y nodo.

Mark Linton acuñó el término “Visitor” en la especificación de Fresco Application Toolkit, de X Consortium [LP93].

PATRONES RELACIONADOS

Composite (151): los visitantes pueden usarse para aplicar una operación sobre una estructura de objetos definida por el patrón Composite.

Interpreter (225): se puede aplicar el patrón Visitor para llevar a cabo la interpretación.

DISCUSIÓN ACERCA DE LOS PATRONES DE COMPORTAMIENTO

ENCAPSULAR LO QUE VARÍA

Encapsular aquello que puede variar es el tema de muchos patrones de comportamiento. Cuando un determinado aspecto de un programa cambia con frecuencia, estos patrones definen un objeto que encapsula dicho aspecto. De esa manera, otras partes del programa pueden colaborar con el objeto siempre que dependan de ese aspecto. Los patrones normalmente definen una clase abstracta que describe el objeto encapsulado, y el patrón toma su nombre de ese objeto[64]. Por ejemplo,

- un objeto Estrategia encapsula un algoritmo (Strategy (289)),
- un objeto Estado encapsula un comportamiento dependiente del estado (State (279)),
- un objeto Mediador encapsula el protocolo entre objetos (Mediator (251)), y
- un objeto Iterador encapsula el modo en que se accede y se recorren los componentes de un objeto agregado (Iterator (237)).

Estos patrones describen aspectos de un programa que es probable que cambien. La mayoría de los patrones tienen dos tipos de objetos: el nuevo objeto que encapsula el aspecto y el objeto existente que usa el nuevo objeto creado. Normalmente, si no fuera por el patrón, la funcionalidad de los nuevos objetos sería una parte integral de los existentes. Por ejemplo, el código de una Estrategia probablemente estaría ligado al Contexto de la estrategia, y el código de un Estado se encontraría implementado directamente en el Contexto del estado.

Pero no todos los patrones de comportamiento de objetos dividen así la funcionalidad. Por ejemplo, el patrón Chain of

Responsibility (205) trata con un número indeterminado de objetos (una cadena), cada uno de los cuales puede que ya exista en el sistema.

El patrón Chain of Responsibility muestra otra diferencia entre los patrones de comportamiento: no todos definen relaciones de comunicación estáticas entre las clases. El patrón Chain of Responsibility describe el modo de comunicación entre un número indefinido de objetos. Otros patrones usan objetos que se pasan como argumentos.

OBJETOS COMO ARGUMENTOS

Varios patrones introducen un objeto que *siempre* se usa como argumento. Uno de ellos es el Visitor (305). Un objeto Visitante es el argumento de una operación polimórfica Aceptar del objeto que visita. El visitante nunca se considera parte de estos objetos, incluso aunque la alternativa convencional al patrón consiste en distribuir el código del Visitante entre las clases de la estructura de objetos.

Otros patrones definen objetos que actúan como elementos mágicos que se pasan de un lado a otro y que más tarde pueden ser invocados. Tanto el patrón Command (215) como el Memento (261) entran en esta categoría. En el Command, el elemento representa una petición; en el Memento, representa el estado interno de un objeto en un momento concreto. En ambos casos, el elemento puede tener una representación interna compleja, pero los clientes nunca llegan a percibirla. No obstante, incluso aquí hay diferencias. El polimorfismo es importante en el patrón Command, ya que ejecutar el objeto Orden es una operación polimórfica. Por el contrario, la interfaz del Memento es tan limitada que éste sólo puede pasarse como un valor. Por tanto, es probable que no presente ninguna operación polimórfica a sus clientes.

LA COMUNICACIÓN, ¿DEBERÍA ESTAR ENCAPSULADA O DISTRIBUIDA?

El Mediator (251) y el Observer (269) son patrones rivales. La diferencia entre ellos es que el patrón Observer distribuye la comunicación introduciendo objetos Observador y Sujeto, mientras que un objeto Mediador encapsula la comunicación entre otros

objetos.

En el patrón Observer, no hay ningún objeto individual que encapsule una restricción. En vez de eso, el Observador y el Sujeto deben cooperar para mantener la restricción. Los patrones de comunicación se determinan por el modo en que se interconectan los observadores y los sujetos: un sujeto individual normalmente tiene muchos observadores, y a veces el observador de un sujeto es un sujeto de otro observador. El patrón Mediator centraliza más que distribuye, ubicando en el mediador la responsabilidad de mantener la restricción.

Hemos descubierto que es más fácil hacer Observadores y Sujetos reutilizables que hacer Mediadores reutilizables. El patrón Observer promueve la separación y bajo acoplamiento entre el Observador y el Sujeto, lo que conduce a clases de grano más fino que son más aptas para ser reutilizadas.

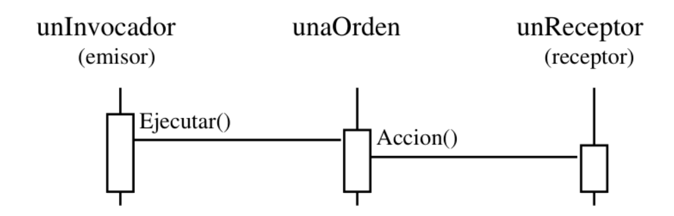
Por otro lado, es más fácil entender el flujo de comunicación en el Mediator que en el Observer. Los observadores y sujetos suelen conectarse nada más crearse, y es difícil ver más tarde en el programa cómo están conectados. Si conocemos el patrón Observen debemos entender que el modo en que se conectan los observadores y los sujetos es importante, y también sabremos qué conexiones buscar. Sin embargo, la indirección introducida por este patrón sigue haciendo que el sistema sea difícil de entender.

Los observadores pueden parametrizarse en Smalltalk con mensajes para acceder al estado del sujeto, lo que los hace más reutilizables de lo que lo son en C++. Esto hace que en Smalltalk sea más atractivo el Observer que el Mediator. De ahí que un programador de Smalltalk generalmente use el Observer donde un programador de C++ usaría un Mediator.

DESACOPLAR EMISORES Y RECEPTORES

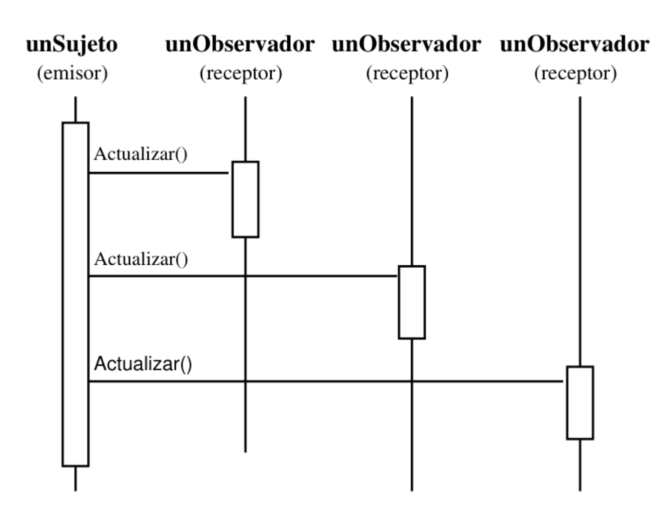
Cuando los objetos que colaboran se refieren unos a otros explícitamente, se vuelven dependientes unos de otros, y eso puede tener un impacto adverso sobre la división en capas y la reutilización de un sistema. Los patrones Command, Observer, Mediator y Chain of Responsibility tratan el problema de cómo desacoplar emisores y receptores, cada uno con sus ventajas e inconvenientes.

El patrón Command permite el desacoplamiento usando un objeto Orden que define un enlace entre un emisor y un receptor:



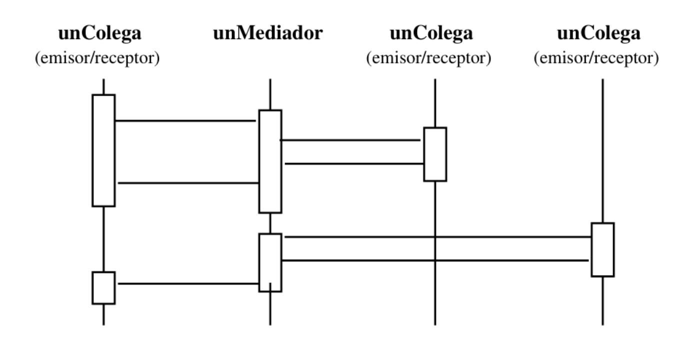
El objeto Orden proporciona una interfaz simple para emitir la petición (es decir, la operación Ejecutar). Definir la conexión emisor-receptor en un objeto aparte permite que el emisor funcione con diferentes receptores. Gracias a mantener al emisor desacoplado de los receptores es más fácil reutilizar los emisores. Más aún, es posible reutilizar el objeto Orden para parametrizar un receptor con diferentes emisores. El patrón Command necesita que haya una subclase por cada conexión emisor-receptor, si bien el patrón describe técnicas de implementación que evitan la herencia.

El patrón Observer desacopla a los emisores (sujetos) de los receptores (observadores) definiendo una interfaz para indicar cambios en los sujetos. Observer define un enlace más débil que Command entre el emisor y el receptor, ya que un sujeto puede tener múltiples observadores, cuyo número puede variar en tiempo de ejecución.



Las interfaces Sujeto y Observador del patrón Observer están diseñadas para posibles cambios en la comunicación. Por tanto, el patrón Observer es mejor para desacoplar objetos cuando hay dependencias de datos entre ellos.

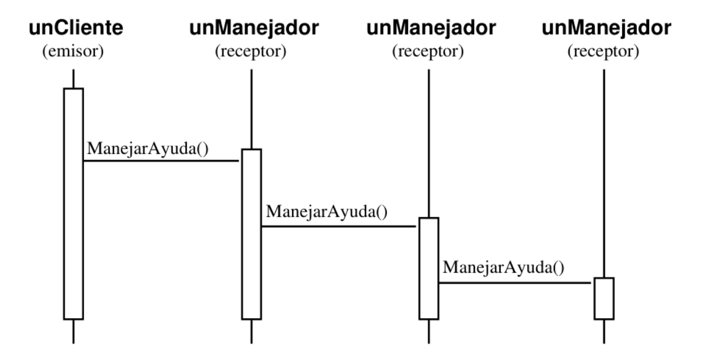
El patrón Mediator desacopla los objetos haciendo que se refieran unos a otros indirectamente, a través del objeto Mediador.



Un objeto Mediador encamina peticiones entre objetos Colega, y centraliza su comunicación. En consecuencia, los colegas sólo pueden hablar entre sí a través de la interfaz del Mediador. Dado que dicha interfaz es fija, el Mediador podría tener que implementar su propio mecanismo de despacho para una flexibilidad añadida. Las peticiones pueden ser codificadas junto con sus argumentos de tal forma que los compañeros pueden solicitar un conjunto de peticiones ilimitado.

El patrón Mediator puede reducir la herencia en un sistema, al centralizar el comportamiento de comunicación en una clase en vez de distribuirlo entre las subclases. Sin embargo, los esquemas de despacho *ad hoc* suelen disminuir la seguridad de tipos.

Por último, el patrón Chain of Responsibility desacopla al emisor del receptor pasando la petición a lo largo de una cadena de receptores potenciales:



Puesto que la interfaz entre emisores y receptores es fija, la Cadena de Responsabilidad también puede requerir un esquema de despacho personalizado. Por tanto, tiene los mismos inconvenientes de seguridad de tipos que el Mediator. La Cadena de Responsabilidad es un buen modo de desacoplar el emisor y el receptor en caso de que la cadena ya forme parte de la estructura del sistema y haya uno o varios objetos capaces de manejar la petición. Y es más, el patrón ofrece una flexibilidad añadida en el sentido de que la cadena puede cambiarse o ampliarse fácilmente.

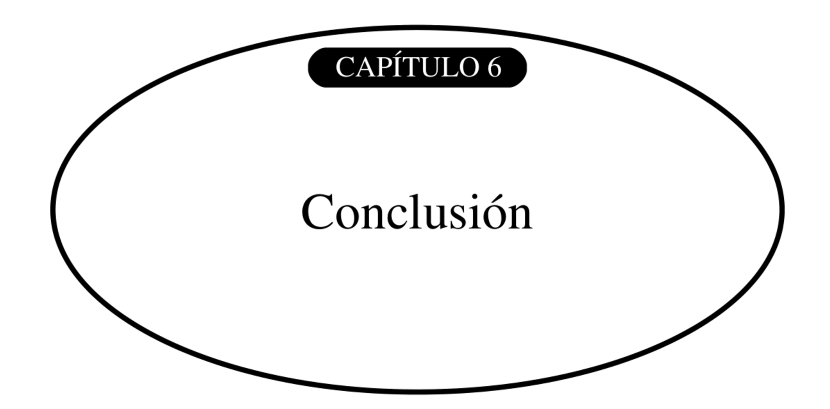
RESUMEN

Con unas pocas excepciones, los patrones de diseño de comportamiento se complementan y se refuerzan entre sí. Una clase de una cadena de responsabilidad, por ejemplo, probablemente incluya al menos una aplicación del patrón Template Method (299). El método de plantilla puede usar operaciones primitivas para determinar si el objeto debería manejar la petición y para elegir el objeto al cual reenviarla. La cadena puede usar el patrón Command para representar peticiones como objetos. El Interpreter (225) puede usar el patrón State para definir contextos para el análisis. Un iterador puede recorrer un agregado, y un visitante puede aplicar una operación a cada elemento del agregado.

Los patrones de comportamiento funcionan bien con otros patrones también. Por ejemplo, un sistema que usa el patrón Composite (151) podría usar un visitante para realizar operaciones sobre los componentes de la composición. Podría usar una cadena de responsabilidad para que los componentes puedan acceder a las propiedades globales a través de su padre. También podría usar el

Decorator (161) para redefinir estas propiedades sobre partes de la composición. O el patrón Observer para ligar una estructura a la otra, y el patrón State para que un componente pueda cambiar su comportamiento cuando cambia su estado. La propia composición podría ser creada usando el enfoque del patrón Builder (89) y se podría tratar como un Prototype (109) por otras partes del sistema.

Los sistemas orientados a objetos bien diseñados no son más que eso —tienen múltiples patrones incrustados en ellos—, pero no necesariamente porque sus diseñadores los hayan pensado en esos términos. La composición de *patrones* en vez de la de clases y objetos nos permite lograr la misma sinergia más fácilmente.



CAPÍTULO 6

Conclusión

CONTENIDO DEL CAPÍTULO

Es posible argüir que este libro no ha logrado gran cosa. Después de todo, no presenta ningún algoritmo o técnica de programación que no se haya usado anteriormente. Tampoco proporciona un método riguroso para el diseño de sistemas, ni desarrolla una nueva teoría de diseño —simplemente, documenta diseños existentes—. Podría concluirse, tal vez, que es una razonable guía de aprendizaje, pero que ciertamente no ofrece gran cosa a un diseñador orientado a objetos experimentado.

Esperamos que no piense así. Catalogar los patrones de diseño es importante. Nos da nombres y definiciones estándar de las técnicas que usamos. Si no estudiáramos los patrones de diseño de software no podríamos mejorarlos, y sería difícil presentar otros nuevos.

Este libro es sólo un comienzo. Contiene algunos de los patrones de diseño más comunes que utilizan los diseñadores orientados a objetos, y que sin embargo la gente conoce y aprende sólo de oídas o estudiando los sistemas existentes. Las versiones preliminares del libro indujeron a otra gente a anotar los patrones de diseño que usaban, y todavía debería inducir más a ello en su nueva forma. Esperamos que esto marque el principio de un movimiento de documentación de la experiencia de los profesionales del software.

Este capítulo discute el impacto que creemos que tendrán los patrones de diseño, cómo se relacionan con otros trabajos de diseño y cómo puede uno involucrarse en la tarea de encontrar y catalogar patrones.

6.1. QUÉ ESPERAR DE LOS PATRONES DE DISEÑO

Los patrones de diseño de este libro pueden afectar de varias formas al modo en que diseñamos software orientado a objetos,

basándonos en nuestra experiencia diaria con ellos.

UN VOCABULARIO DE DISEÑO COMÚN

Los estudios de programadores expertos de lenguajes convencionales muestran que el conocimiento y la experiencia no se organizan simplemente en torno a la sintaxis, sino en estructuras conceptuales mayores, tales como algoritmos, estructuras de datos y modismos [AS85, Cop92, Cur89, SS86], así como en maneras de lograr un determinado objetivo [SE84]. Los diseñadores probablemente no piensan en la notación que están usando para documentar su diseño, sino que intentan comparar la situación actual de diseño con planos, algoritmos, estructuras de datos y modismos que han aprendido en el pasado.

Los informáticos nominan y catalogan las estructuras de datos y algoritmos, pero no suelen dar nombre a otros tipos de patrones. Los patrones de diseño proporcionan un vocabulario común que los diseñadores usan para comunicar, documentar y explorar alternativas de diseño. Los patrones de diseño hacen que un sistema parezca menos complejo, permitiéndonos hablar de él con un mayor nivel de abstracción del que permite una notación de diseño o un lenguaje de programación. Los patrones de diseño elevan el nivel en el que diseñamos y discutimos diseños con nuestros colegas.

Una vez que haya absorbido los patrones de diseño de este libro, su vocabulario de diseño cambiará casi con toda seguridad. Hablará directamente en términos de los nombres de los patrones de diseño. Se oirá a sí mismo decir cosas como “aquí deberíamos usar el patrón Observer”, o “extraigamos una estrategia de estas clases”.

UNA AYUDA PARA LA DOCUMENTACIÓN Y EL APRENDIZAJE

Conocer los patrones de diseño de este libro facilita la comprensión de los sistemas existentes. La mayoría de los grandes sistemas orientados a objetos usan estos patrones de diseño. La gente, cuando está aprendiendo programación orientada a objetos, muchas veces se queja de que los sistemas con los que trabaja usan la herencia de una forma enrevesada, y de que es difícil seguir el flujo de control. En gran parte esto es debido a que no comprenden los patrones de diseño del sistema. Aprender estos patrones de diseño

le ayudará a comprender los sistemas software existentes.

Estos patrones de diseño también pueden convertirle en un diseñador mejor. Si trabaja con sistemas orientados a objetos lo suficientemente grandes, probablemente aprenda estos patrones por sí mismo. Pero leer el libro le ayudará a aprender mucho más deprisa. Aprender estos patrones ayudará a un novato a comportarse más como un experto.

Más aún, describir un sistema en términos de los patrones que usa hará que éste sea mucho más fácil de entender. De otro modo, la gente tendrá que hacer ingeniería inversa sobre el diseño para descubrir los patrones que usa. Tener un vocabulario común significa que no tenemos que describir el patrón de diseño entero; podemos simplemente nombrarlo y confiar en que quien lo lea lo conozca. Un lector que no conoce los patrones tendrá que buscarlos en primer lugar, pero eso sigue siendo más fácil que hacer ingeniería inversa.

Nosotros usamos estos patrones en nuestros propios diseños y pensamos que tienen un valor incalculable. No obstante, usamos los patrones de forma simplista y discutible. Los usamos para elegir nombres para las clases, para pensaren un buen diseño y enseñarlo, y para describir diseños en términos de la secuencia de patrones que hemos aplicado [BJ94]. Es fácil imaginarse formas más sofisticadas de usar los patrones, como herramientas CASE o documentos hipertexto basados en patrones. Pero los patrones son una gran ayuda incluso sin herramientas sofisticadas.

UN COMPLEMENTO DE LOS MÉTODOS EXISTENTES

Los métodos de diseño orientados a objetos están pensados para promover el buen diseño, para enseñar a los nuevos diseñadores cómo diseñar bien y para estandarizar el modo en que se desarrollan los diseños. Un método de diseño generalmente define un conjunto de notaciones (normalmente gráficas) para modelar varios aspectos de un diseño, junto con un conjunto de reglas que gobiernan cómo y cuándo usar cada notación. Los métodos de diseño normalmente describen los problemas que tienen lugar en un diseño, cómo resolverlos y cómo evaluar el diseño. Pero se han mostrado incapaces de representar la experiencia de los diseñadores expertos.

Creemos que nuestros patrones de diseño son una muestra importante de lo que hemos estado echando de menos en los métodos de diseño. Los patrones de diseño muestran cómo usar técnicas primitivas como objetos, herencia y polimorfismo. Muestran cómo parametrizar un sistema con un algoritmo, un comportamiento, un estado o con la clase de objetos que debe crear. Los patrones de diseño proporcionan una forma de describir mejor "el porqué" de un diseño, en vez de limitarse a registrar los resultados de nuestras decisiones. Las secciones de Aplicabilidad, Consecuencias e Implementación de los patrones de diseño le ayudarán en las decisiones que tenga que tomar.

Los patrones de diseño son especialmente útiles a la hora de pasar de un modelo de análisis a un modelo de implementación. A pesar de que muchas afirmaciones prometen una suave transición del análisis orientado a objetos al diseño, en la práctica la transición es cualquier cosa menos suave. Un diseño flexible y reutilizable contendrá objetos que no están en el modelo de análisis. El lenguaje de programación y las bibliotecas de clases utilizados afectan al diseño. Los modelos de análisis muchas veces deben ser rediseñados para hacerlos más reutilizables. Muchos de los patrones de diseño del catálogo se enfrentan a estos problemas, motivo por el que los denominamos patrones *de diseño*.

Un método de diseño completo necesita patrones de más tipos que los de diseño. También puede haber patrones de análisis, patrones de diseño de interfaces de usuario o patrones de ajuste del rendimiento. Pero los patrones de diseño son una parte esencial, que ha sido obviada hasta ahora.

UN OBJETIVO PARA LA REFACTORIZACIÓN

Uno de los problemas del desarrollo de software reutilizable es que a menudo tiene que ser reorganizado o **refactorizado** [OJ90]. Los patrones de diseño ayudan a determinar cómo reorganizar un diseño, y pueden reducir la cantidad de refactorización que es necesario hacer luego.

El ciclo de vida del software orientado a objetos tiene varias fases. Brian Foote identifica éstas como las fases de **prototipado, expansión** y **consolidación** [Foo92].

La fase de prototipado es una oleada de actividad en la que el software ve la luz a través del prototipado rápido y de los cambios increméntales, hasta que se descubren un conjunto de requisitos iniciales y alcanza la adolescencia. En este punto, el software normalmente consiste en jerarquías de clases que reflejan fielmente las entidades del dominio del problema inicial. El principal tipo de reutilización es de caja blanca mediante la herencia.

Una vez que el software ha alcanzado su adolescencia y se pone en servicio, su evolución está gobernada por dos necesidades contradictorias: (1) el software debe satisfacer más requisitos, y (2) el software debe ser más reutilizable. Nuevos requisitos añaden nuevas clases y operaciones, y tal vez jerarquías de clases completas. El software cambia hacia una fase de expansión para descubrir nuevos requisitos. Esto, no obstante, no puede continuar durante largo tiempo. Al final el software se volverá demasiado inflexible para permitir más cambios. Las jerarquías de clases ya no se corresponderán con ningún dominio de problema. En vez de eso, reflejarán muchos dominios de problemas, y las clases definirán muchas operaciones y variables de instancia sin relación entre sí.

Para seguir evolucionando, el software debe ser reorganizado en un proceso conocido como *refactorización*. Ésta es la fase en la que suelen aparecer los frameworks. Refactorizar implica desbrozar las clases en componentes de propósito general y especial, moviendo operaciones hacia arriba y hacia abajo en la jerarquía de clases, y racionalizando las interfaces de las clases. Esta fase de consolidación produce muchos nuevos tipos de objetos, a menudo mediante descomposición de los objetos existentes, y usando composición de objetos en vez de la herencia. Por tanto, la reutilización de caja negra sustituye a la de caja blanca. La necesidad constante de satisfacer más requisitos, junto con la necesidad de más reutilización lleva al software orientado a objetos a través de repetidas fases de expansión y consolidación —expansión a medida que se satisfacen nuevos requisitos, y consolidación a medida que el software se vuelve más general—.

expansión

más requisitos

más reutilización

prototipado

consolidación

Este ciclo es inevitable. Pero los buenos diseñadores están pendientes de los cambios que pueden dar lugar a refactorizaciones. Los buenos diseñadores también conocen estructuras de clases y objetos que pueden ayudarles a evitar refactorizaciones —sus diseños son más robustos frente a los cambios de requisitos—. Un análisis de requisitos concienzudo pondrá de manifiesto aquellos requisitos que es probable que cambien durante la vida del software, y un buen diseño será robusto frente a dichos cambios.

Nuestros patrones de diseño reflejan muchas de las estructuras que resultan de la refactorización. Usando estos patrones en las fases tempranas del diseño se previenen posteriores refactorizaciones. Pero incluso aunque no veamos cómo aplicar un patrón hasta que hayamos construido nuestro sistema, el patrón todavía puede mostrarnos cómo cambiarlo. Los patrones de diseño nos proporcionan así objetivos para nuestras refactorizaciones.

6.2. UNA BREVE HISTORIA

El catálogo comenzó como parte de la tesis doctoral de Erich [Gam9l. Gam92]. Cerca de la mitad de los patrones actuales estaban en su tesis. Para el OOPSLA '91 había oficialmente un catálogo independiente, y Richard se unió a Erich para trabajar en él. John comenzó a trabajar en él enseguida. Para el OOPSLA '92. Ralph se había unido al grupo. Trabajó duró para ajustar el catálogo para su publicación en ECOOP '93, pero pronto nos dimos cuenta de que un artículo de 90 páginas no iba a ser aceptado. De modo que

resumimos el catálogo y enviamos el resumen, que fue aceptado. Al poco, decidimos convertir el catálogo en un libro.

Nuestros nombres de los patrones han cambiado algo desde entonces. “Wrapper” (envoltorio) se convirtió en “Decorator” (decorador). “Glue” (pegamento) se convirtió en “Facade” (fachada). “Solitaire” (solitario) en “Singleton” (único), y “Walker” (caminante) en “Visitor” (visitante). Se quitaron un par de patrones porque no parecían lo suficientemente importantes. Pero, aparte de eso, el conjunto de patrones del catálogo ha cambiado poco desde finales de 1992. Los patrones en sí, no obstante, han evolucionado tremendamente.

De hecho, darse cuenta de que algo es un patrón es la parte fácil. Todos nosotros trabajamos de forma activa en la construcción de sistemas orientados a objetos, y hemos descubierto que es fácil reconocer a los patrones cuando se miran bastantes sistemas. Pero *encontrar* patrones es mucho más fácil que *describirlos*.

Si construye sistemas y a continuación reflexiona sobre lo que ha construido, verá patrones en lo que hace. Pero es difícil describir los patrones para que otra gente que no los conoce los comprenda y se dé cuenta de por qué son importantes. Los expertos reconocieron inmediatamente el valor del catálogo en sus etapas iniciales. Pero los únicos que podrían entender los patrones eran aquellos que ya los habían usado.

Puesto que uno de los principales propósitos del libro era enseñar diseño orientado a objetos a los nuevos diseñadores, sabíamos que teníamos que mejorar el catálogo. Ampliamos el tamaño medio de un patrón de menos de 2 a más de 10 páginas, incluyendo un detallado ejemplo de motivación y código de ejemplo. También comenzamos a examinar las ventajas e inconvenientes y las distintas formas de implementar el patrón. Esto hizo que los patrones fuesen más fáciles de entender.

Otro cambio importante del último año ha sido dar más importancia al problema que es resuelto por un patrón. Lo más fácil es ver a un patrón como una solución, como una técnica que puede adaptarse y reutilizarse. Es más difícil ver cuándo resulta *apropiado* —caracterizar los problemas que resuelve y el contexto en el que constituye la mejor solución—. En general, es más fácil ver *qué* hace algo que saber *por qué*, y el “porqué” de un patrón es el

problema que resuelve. Conocer el propósito de un patrón también es importante, porque nos ayuda a elegir los patrones a aplicar. El autor de un patrón debe establecer y caracterizar el problema que resuelve el patrón, incluso aunque tenga que hacerlo después de haber descubierto la solución.

6.3. LA COMUNIDAD DE PATRONES

Nosotros no somos los únicos interesados en escribir libros que cataloguen los patrones que utilizan los expertos. Formamos parte de una comunidad mayor interesada en los patrones en general, y en los patrones de software en particular. Christopher Alexander es el arquitecto que primero estudió los patrones en los edificios y comunidades y que desarrolló un "lenguaje de patrones" para generarlos. Su trabajo nos ha inspirado una y otra vez. Por tanto, es adecuado y merece la pena comparar su trabajo con el nuestro. A continuación veremos el trabajo de otros en patrones de software.

LOS LENGUAJES DE PATRONES DE ALEXANDER

Nuestro trabajo se parece al de Alexander en muchos aspectos. Ambos están basados en observar los sistemas existentes y buscar patrones en ellos. Ambos tienen plantillas para describir patrones (si bien nuestras plantillas son bastante diferentes). Ambos se basan en el lenguaje natural y en montones de ejemplos para describir los patrones, en vez de en lenguajes formales, y ambos dan las razones de cada patrón.

Pero hay otros muchos aspectos en los que nuestros trabajos difieren:

1. La gente ha estado haciendo casas durante miles de años, y hay muchos ejemplos clásicos a los que acudir. Hemos estado haciendo sistemas software durante un periodo de tiempo relativamente corto, y hay pocos que puedan considerarse clásicos.
2. Alexander da un orden en el que deberían utilizarse sus

patrones; nosotros no.

3. Los patrones de Alexander hacen hincapié en el problema que solucionan, mientras que los patrones de diseño describen con más detalle la solución.
4. Alexander afirma que sus patrones generarán edificios completos. Nosotros no afirmamos que nuestros programas generarán programas completos.

Cuando Alexander afirma que se puede diseñar una casa simplemente aplicando sus patrones uno detrás de otro, tiene metas similares a las de esos metodólogos del diseño orientado a objeto que dan recetas de diseño. Alexander no niega la necesidad de la creatividad; algunos de sus patrones requieren comprender los hábitos de vida de la gente que usará las casas, y su creencia en la "poesía" del diseño implica un nivel de experiencia más allá del lenguaje de patrones en sí[65]. Pero su descripción de cómo los patrones generan diseño implica que un lenguaje de patrones puede hacer que el proceso de diseño sea determinista y repetible.

El punto de vista de Alexander nos ha ayudado a centramos en las ventajas e inconvenientes de los diseños —las diferentes "fuerzas" que ayudan a dar forma a un diseño—. Su influencia nos hizo trabajar más duro para comprender la aplicabilidad y consecuencias de nuestros patrones. También evitó que nos preocupáramos de definir una representación formal para los patrones. Aunque dicha representación podría hacer posible automatizar los patrones, en este punto es más importante explorar el espacio de los patrones de diseño que formalizarlo.

Desde el punto de vista de Alexander, los patrones de este libro no constituyen un lenguaje. Dada la gran cantidad de sistemas software que se construyen, es difícil imaginarse cómo podríamos proporcionar un conjunto "completo" de patrones, que ofreciese instrucciones paso a paso para diseñar una aplicación. Podemos hacerlo para ciertas clases de aplicaciones, tales como escribir informes o hacer un sistema de entrada de formularios. Pero nuestro catálogo tan sólo es una colección de patrones relacionados; no podemos pretender que sea un lenguaje de patrones.

De hecho, pensamos que es poco probable que haya *nunca* un lenguaje de patrones completo para el software. No obstante, es

ciertamente posible hacer uno que sea *más* completo. Se podrían haber añadido los frameworks y cómo usarlos [Joh92], patrones para el diseño de interfaces de usuario [BJ94], patrones de análisis [Coa92] y todos los otros aspectos del desarrollo de software. Los patrones de diseño son sólo una parte de un lenguaje mayor de patrones de software.

PATRONES DE SOFTWARE

Nuestra primera experiencia colectiva en el estudio de la arquitectura del software fue en un taller de OOPSLA ‘91 conducido por Bruce Anderson. El taller estaba dedicado a desarrollar un manual para arquitectos de software (a juzgar por este libro, sospechamos que “enciclopedia de arquitectura” sería un nombre más apropiado que “manual de arquitectura”). Ese primer taller llevó a una serie de encuentros, el más reciente de los cuales fue la primera conferencia de Lenguajes de Patrones de Programas (*Pattern Languages of Programs*) celebrada en agosto de 1994. Esto ha creado una comunidad de personas interesadas en documentar la experiencia en el software.

Por supuesto, otros ya habían tenido este objetivo. El *The Art of Computer Programming* de Donald Knuth [Knu73] fue uno de los primeros intentos de catalogar el conocimiento en el software, aunque él se centró en describir algoritmos. Incluso así, la tarea se evidenció demasiado grande como para ser terminada. Las series de *Graphics Gems* [Gla90, Arv91, Kir92] son otro catálogo de conocimiento de diseño, aunque también tiende a centrarse en algoritmos. El programa de Arquitectura de Software Específica del Dominio, patrocinado por el Departamento de Defensa de los Estados Unidos [GM92], se centra en recoger información arquitectónica. La comunidad de ingeniería del software basada en el conocimiento trata de representar el conocimiento relativo al software en general. Hay muchos otros grupos con objetivos al menos similares al nuestro.

Advanced C++." *Programming Styles and idioms*, de James Coplien [Cop92] también nos ha influido. Los patrones de este libro tienden a ser más específicos de C++ que los nuestros, y su libro también contiene muchos patrones de más bajo nivel. Pero hay

cosas en común, como señalamos en nuestros patrones. Jim ha sido un miembro activo de la comunidad de patrones. Actualmente está trabajando en patrones que describen los roles de las personas en las organizaciones de desarrollo de software.

Hay otros muchos sitios en los que se pueden encontrar descripciones de patrones. Kent Beck fue una de las primeras personas de la comunidad del software que defendió el trabajo de Christopher Alexander. En 1993 comenzó a escribir una columna en *The Smalltalk Report* sobre patrones de Smalltalk. Peter Coad también había pasado cierto tiempo coleccionando patrones. Su artículo sobre patrones nos parece que contiene sobre todo patrones de análisis [Coa92j; no hemos visto sus últimos patrones, aunque sabemos que sigue trabajando en ellos. Hemos oído hablar de varios libros sobre patrones que están escribiéndose, pero tampoco hemos visto ninguno. Todo lo que podemos hacer es decir que están en camino. Uno de estos libros será de la conferencia de Lenguajes de Patrones de Programas.

6.4. UNA INVITACIÓN

¿Qué puede hacer si está interesado en los patrones? En primer lugar, úselos y busque otros patrones que se ajusten a su diseño. En los próximos años aparecerán un montón de libros y artículos acerca de patrones, de modo que habrá muchas fuentes de nuevos patrones. Desarrolle su vocabulario de patrones y úselo. Úselo cuando hable con otras personas sobre sus diseños. Úselo cuando piense y escriba sobre ellos.

En segundo lugar, sea un consumidor crítico. El catálogo de patrones de diseño es resultado de un duro trabajo, no sólo nuestro, sino de docenas de revisores que nos dieron su opinión. Si encuentra un problema o cree que es necesaria más explicación, póngase en contacto con nosotros. Lo mismo vale para cualquier otro catálogo de patrones: ¡deles su opinión a los autores! Una de las mejores cosas de los patrones es que apartan las decisiones de diseño del reino de la mera intuición. Permiten que los autores sean explícitos sobre las ventajas e inconvenientes que proporcionan, lo

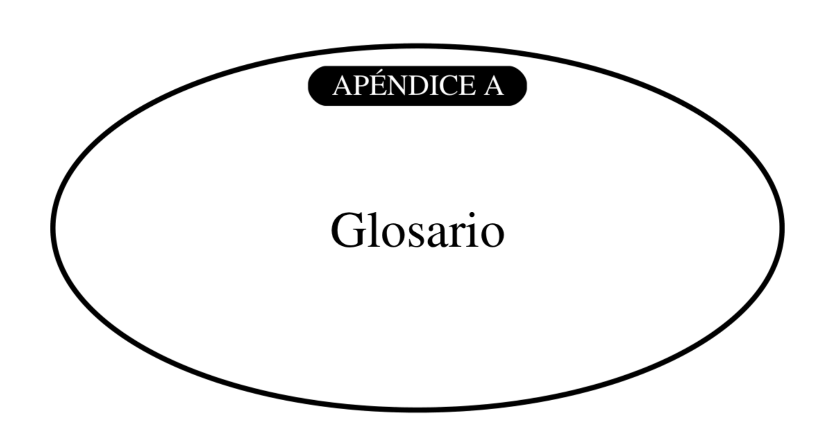
que facilita ver los fallos de sus patrones y discutir con ellos. Aprovéchese de ello.

En tercer lugar, observe qué patrones usa y anótelos. Haga de ellos parte de su documentación. Muéstrelos a otras personas. No necesita estar en un laboratorio de investigación para descubrir patrones. De hecho, encontrar patrones relevantes es prácticamente imposible si carece de experiencia práctica. Anímese a escribir su propio catálogo de patrones… ¡pero asegúrese de que alguien más le ayuda a darles forma!

6.5. UNA REFLEXIÓN A MODO DE DESPEDIDA

Los mejores diseños usarán muchos patrones de diseño que se imbrican entre sí para producir un mejor todo. Como afirma Christopher Alexander:

> *Es posible hacer edificios enlazando patrones, de un modo poco preciso. Un edificio así construido es una mezcolanza de patrones. No es denso. No es profundo. Pero también es posible juntar patrones de modo que muchos patrones se solapen en un mismo espacio físico: el edificio es muy denso: tiene muchos significados representados en un espacio reducido: y a través de esa densidad, se hace profundo.*
>
> *A Pattern Language [AIX+77, página xli]*


APÉNDICE A

Glosario

acoplamiento

El grado en que los componentes software dependen unos de oíros.

acoplamiento abstracto

Dada una clase *A* que tiene una referencia a una clase abstracta *B*, la clase *A* se dice que tiene un *acoplamiento abstracto a B*. Lo llamamos acoplamiento abstracto porque *A* se refiere al *tipo* de un objeto, no a un objeto concreto.

clase

Una clase define la interfaz de un objeto y su implementación. Especifica la representación interna de un objeto y define las operaciones que éste puede llevar a cabo.

clase abstracta

Una clase cuyo principal propósito es definir una interfaz. Una clase abstracta delega parte de su implementación, o toda, en sus subclases. No se pueden crear instancias de una clase abstracta.

clase mezclable

Una clase diseñada para ser combinada con otras por medio de la herencia. Las clases mezclables suelen ser abstractas.

clase amiga

En C+ +, una clase que tiene los mismos permisos de acceso a las operaciones y datos de la clase que la propia clase.

clase concreta

Una clase que no tiene operaciones abstractas. Se pueden crear instancias de ella.

clase padre
: La clase de la que, hereda otra clase. Tiene como sinónimos superclase (Smalltalk), clase base (C+ +) y clase antecesora.

composición de objetos
: Ensamblar o componer objetos para obtener un comportamiento más complejo.

constructor
: En C+ +, una operación que se invoca automáticamente para inicializar las nuevas instancias

delegación
: Un mecanismo de implementación mediante el cual un objeto redirige o delega una petición a otro objeto. El delegado lleva a cabo la petición en nombre del objeto original.

destructor
: En C+ +, una operación a la que se invoca automáticamente para terminar un objeto que está a punto de ser borrado.

diagrama de clases
: Un diagrama que representa clases, su estructura y operaciones internas, y las relaciones estáticas entre ellas.

diagrama de interacción
: Un diagrama que muestra el finjo de peticiones entre objetos.

diagrama de objetos
: Un diagrama que representa una determinada estructura de objetos en tiempo de ejecución.

encapsulación
: El resultado de ocultar la representación e implementación en un objeto. 1.a representación no es visible y no se puede acceder a ella directamente desde el exterior del objeto. El único modo de acceder a la representación de un objeto y de modificarla es a través de sus operaciones.

enlace dinámico

La asociación en tiempo de ejecución entre una petición a un objeto y una de sus operaciones. En C++, sólo las funciones virtuales puras están enlazadas dinámicamente.

framework

Un conjunto de clases cooperantes que forman un diseño reutilizable para una determinada clase de software. Un framework proporciona tina gula arquitectónica para dividir el diseño en clases abstractas y definir sus responsabilidades y colaboraciones. Un desarrollador adapta cl framework a una aplicación concreta heredando y componiendo instancias de las clases del framework.

herencia

Una relación que define una entidad en términos de otra. La herencia de clases define una nueva clase en términos de una o más clases padre. La nueva dase hereda su interfaz y sus implementaciones de sus padres. La nueva clase se dice que es una subclase o (en C++) una clase derivada. La herencia de clases combina herencia de Interfaces y herencia de implementación. La herencia de interfaces define una nueva interfaz en términos de una o varias interfaces existentes. La herencia de implementación define una nueva implementación en términos de una o varias implementaciones existentes.

herencia privada

En C++, una clase de la que se hereda sólo por su implementación.

interfaz

El conjunto de todas las signaturas definidas por las operaciones de un objeto. La interfaz describe el conjunto de peticiones a las que puede responder un objeto.

metaclase

Las clases son objetos en Smalltalk. Una metaclase es la clase de un objeto clase.

objeto

Una entidad de tiempo de ejecución que empaqueta datos y los procedimientos que operan sobre esos datos.

objeto agregado

Un objeto que se compone de subobjetos. Los subobjetos se denominan las partes, y el agregado es responsable de ellos.

operación

Los datos de un objeto sólo pueden ser manipulados por sus operaciones. Un objeto realiza una operación cuando recibe una petición. En C++, a las operaciones se tas denomina funciones miembro. Smalltalk usa el término método.

operación abstracta

Una operación que declara una signatura pero no la implementa. En C++, una operación abstracta se corresponde con una función miembro virtual pura.

operación de clase

Una operación que pertenece a una clase y no a un objeto individual. En C++, las operaciones de clase se denominan funciones miembro estáticas.

patrón de diseño

Un patrón de diseño nomina, da los motivos y explica sistemáticamente un diseño general que resuelve un problema de diseño recurrente en los sistemas orientados a objetos. Describe el problema, la solución, cuándo aplicar ésta y sus consecuencias. También ofrece trucos de implementación y ejemplos. La solución es una disposición general de clases y objetos que resuelven el problema. Está adaptada e implementada para resolver el problema en un determinado contexto.

petición

Un objeto lleva a cabo una operación cuando recibe la petición correspondiente de otro objeto. Un sinónimo frecuente de petición es mensaje.

polimorfismo

La capacidad de sustituir los objetos que se ajustan a una interfaz por otros en tiempo de ejecución.

protocolo

Extiende el concepto de interfaz para incluir todas las secuencias de peticiones permitidas.

receptor

El objeto destino de una petición.

redefinición

Volver a definir una operación (heredada de una clase padre) en una subclase.

referencia de objetos

Un valor que identifica a otro objeto.

relación de agregación

La relación entre un objeto agregado y sus partes. Una clase define esta relación con sus instancias (objetos agregados).

relación de asociación

Una clase que se refiere a otra clase tiene una asociación con esa clase.

reutilización de caja blanca

Un estilo de reutilización basado en la herencia de clases. Una subclase reutiliza la interfaz e implementación de su clase padre, pero puede acceder a los aspectos privados de su padre.

reutilización de caja negra

Un estilo de reutilización basado en la composición de objetos. Los objetos compuestos no se revelan entre sí sus detalles internos, lo que los hace ser como "cajas negras".

signatura

La signatura de una operación define su nombre, parámetros y tipo de retomo.

subclase
: Una clase que hereda de otra. En C++, una subclase se denomina una clase derivada.

subsistema
: Un grupo independiente de clases que colaboran para llevar a cabo una serie de responsabilidades.

subtipo
: Un tipo es un subtipo de otro si su interfaz contiene a la interfaz de aquél.

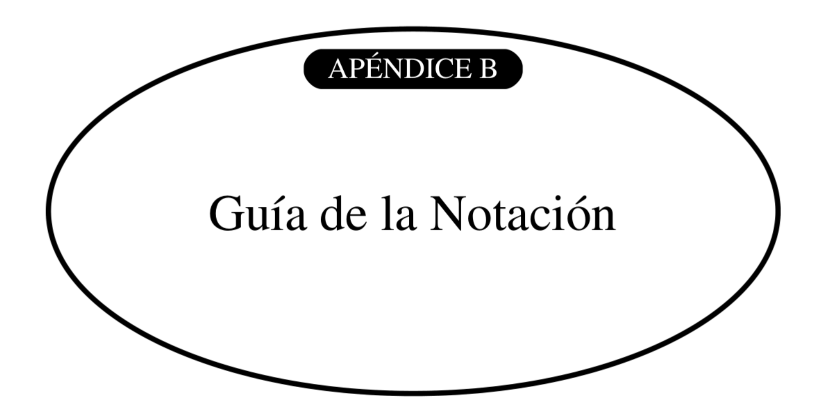
supertipo
: El tipo padre del que hereda otro tipo.

tipo
: El nombre de una determinada interfaz.

tipo parametrizado
: Un tipo que deja sin especificar alguno de sus tipos constituyentes. Los tipos sin especificar se proporcionan como parámetros en el momento de su uso. En C++, los tipos parametrizados se llaman plantillas.

toolkit
: Una colección de clases que proporcionan una funcionalidad útil pero que no definen el diseño de una aplicación.

variable de instancia
: Un elemento de datos que define parte de la representación de un objeto. C++ usa el término **miembro de datos**.



CONTENIDO DEL APÉNDICE

A lo largo del libro se usan diagramas para ilustrar las ideas importantes. Algunos diagramas son informales, como una captura de pantalla de un cuadro de diálogo o un esquema que representa un árbol de objetos. Pero lo que son los patrones de diseño usan notaciones más formales para denotar relaciones e interacciones entre clases y objetos. En este apéndice se describen dichas notaciones en detalle.

Usamos tres notaciones gráficas distintas:

1. Un **diagrama de clases** representa ciases, su estructura y las relaciones estáticas entre ellas.
2. Un **diagrama de objetos** muestra una determinada estructura de objetos en tiempo de ejecución.
3. Un **diagrama de interacción** muestra el flujo de peticiones entre objetos.

Cada patrón de diseño incluye al menos un diagrama de clases. Las otras notaciones se usan cuando son necesarias para completar la descripción. Los diagramas de clases y objetos están basados en OMT (*Object Modeling Technique*, Técnica de Modelado de Objetos) [RBP+91, Rum94] **[66]**. Los diagramas de interacción están tomados de Objectory [JCJ092] y del método de Booch [Boo94]. Estas notaciones están resumidas en la contraportada interior del libro.

B.1. DIAGRAMA DE CLASES

La figura B. la muestra la notación OMT para las clases abstractas y concretas. Una clase se denota por un rectángulo con el nombre de la clase en negrita en la parte superior. Las principales operaciones

de la clase aparecen bajo el nombre de la clase. Las variables de instancia se muestran debajo de las operaciones. La información de tipos es opcional; nosotros usamos el convenio de C++, que sitúa el nombre del tipo antes del nombre de la operación (para significar el tipo de retorno), variable de instancia o parámetro real. Si el tipo aparece en cursiva quiere decir que la operación es abstracta.

En algunos diseños resulta útil ver dónde hacen referencia las clases cliente a las clases Participante. Cuando un patrón incluye una clase Cliente como una de sus participantes (lo que quiere decir que el cliente tiene alguna responsabilidad en el patrón, el Cliente aparece como una clase ordinaria. Un ejemplo de esto lo tenemos en el patrón Flyweight (179). Cuando el patrón no incluye al Cliente como participante (es decir, los clientes no tienen responsabilidades en el patrón), pero a pesar de eso incluirlo en el diagrama ayuda a clarificar el modo en que el patrón interactúa con sus clientes, entonces la clase Cliente se muestra en gris, como en la figura B.1b. Un ejemplo de esto es el patrón Proxy (191). Un Cliente en gris también deja claro que no ha sido omitido accidentalmente de la sección de Participantes.

La figura B.1c muestra varias relaciones entre clases. La notación OMT para la herencia de clases es un triángulo que conecta a una subclase (FormaLinea en la figura) a su clase padre (Forma). Una referencia a un objeto que representa una relación de agregación o parte-todo se indica mediante una flecha con un rombo en su base. La flecha apunta a la clase del agregado (por ejemplo. Forma). Una flecha sin rombo denota asociación (por ejemplo, una FormaLinea tiene una referencia a un objeto Color que puede ser compartido con otras formas). La referencia puede tener un nombre cerca de la base para distinguirla de otras referencias [67].

Figura B.1: Notación de los diagramas de clases

NombreClaseAbstracta

OperacionAbstracta1()
Tipo OperacionAbstracta2()

NombreClaseConcreta

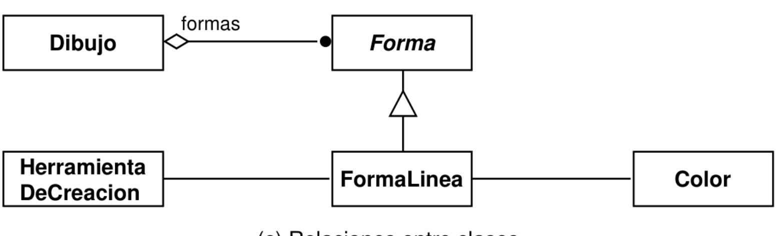
OperacionAbstracta1()
Tipo OperacionAbstracta2()

OperacionAbstracta1()
Tipo OperacionAbstracta2()

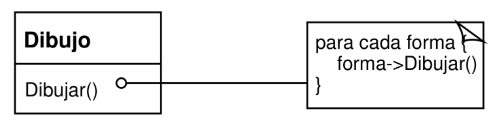
(a) Clases abstractas y concretas



(b) Clases participantes Cliente (izquierda) y Cliente implícito (derecha



(c) Relaciones entre clases



(d) Notación en pseudocódigo

También es útil mostrar qué clases crean instancias de qué otras. Para ello usamos una flecha con la línea punteada, ya que OMT no lo permite. Llamamos a esto la relación “crea”. La flecha apunta a la clase de la que se crea la instancia. En la figura B.1c, HerramientaDeCreacion crea objetos Forma-Linea.

OMT también define un círculo relleno que representa “más de uno”. Cuando el círculo aparece en el extremo de una referencia

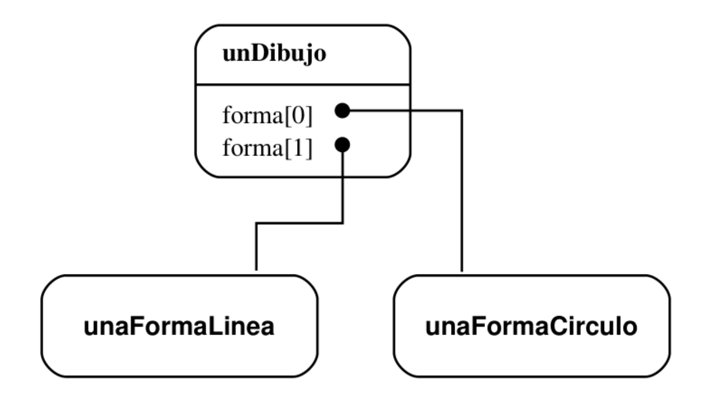
quiere decir que hay muchos objetos referenciados o agregados. La figura B.1c representa un agregado Dibujo que contiene múltiples objetos de tipo Forma.

Por último, hemos aumentado OMT con anotaciones en pseudocódigo que nos permiten esbozar las implementaciones de las operaciones. La figura B.1d muestra la anotación en pseudocódigo para la operación Dibujar de la clase Dibujo.

B.2. DIAGRAMA DE OBJETOS

Un diagrama de objetos muestra exclusivamente instancias. Representa una instantánea de los objetos de un patrón de diseño. Los objetos tienen por nombre "un*Algo*" donde *Algo* es la clase del objeto. Nuestro símbolo de un objeto (modificado ligeramente del estándar OMT) es un rectángulo con los bordes redondeados y una línea que separa el nombre del objeto de las referencias a otros objetos. Las flechas indican el objeto referenciado. La figura B.2 muestra un ejemplo.

Figura B.2: Notación de los diagramas de objetos

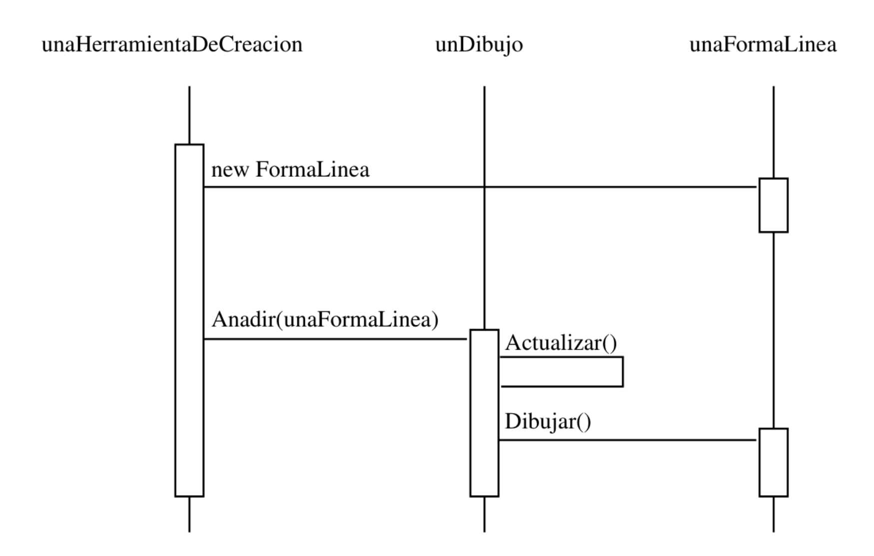


B.3. DIAGRAMA DE INTERACCIÓN

Un diagrama de interacción muestra el orden en que se ejecutan las peticiones entre objetos. La figura B.3 es un diagrama de interacción que muestra cómo se añade una forma a un dibujo.

En un diagrama de interacción, el tiempo fluye de arriba a abajo. Una línea vertical indica el tiempo de vida de un determinado objeto. La convención de nominación de objetos es la misma que la de los diagramas de objetos —el nombre de la clase con los artículos indeterminados "un" o "una" como prefijo (por ejemplo, unaForma)—. Si no se crea ninguna instancia del objeto hasta un tiempo después del instante inicial representado en el diagrama, entonces su línea vertical aparece punteada hasta el momento de la creación.

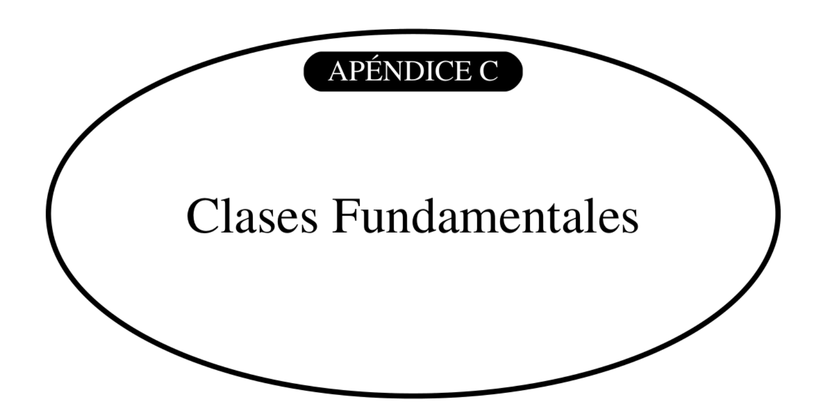
Figura B.3: Notación de los diagramas de interacción



Un rectángulo vertical indica que un objeto está activo; es decir, que está procesando una petición. La operación puede enviar peticiones a otros objetos; esto se indica con una flecha horizontal que apunta al objeto receptor. El nombre de la petición se muestra encima de la flecha. Una petición de creación de un objeto se indica con una flecha punteada. Una petición al mismo objeto emisor se representa con una flecha hacia sí mismo.

La figura B.3 muestra que la primera petición procede de unaHerramientaDeCreacion para crear unaFormaLinea. A

continuación, se añade unaFormaLinea a unDibujo, lo que hace que unDibujo se envíe a sí mismo una petición Actualizar. Nótese que unDibujo envía una petición Dibujar a unaFormaLinea como parte de la operación Actualizar.



CONTENIDO DEL APÉNDICE

Este apéndice documenta las clases fundamentales que usamos en el código de ejemplo C++ de varios patrones de diseño. Las hemos mantenido intencionadamente simples y mínimas. Describiremos las clases siguientes:

- Lista, una lista ordenada de objetos.
- Iterador, la interfaz para acceder en secuencia a los objetos de un agregado.
- IteradorLista, un iterador para recorrer una Lista.
- Punto, un punto de dos dimensiones.
- Rect, un rectángulo.

Algunos de los tipos estándar más recientes de C++ puede que no estén disponibles en todos los compiladores. En concreto, si nuestro compilador no define bool, podemos definirlo manualmente como sigue:

```
typedef int bool;
const int true = 1;
const int false = 0;
```

C.1. LISTA

La clase de plantilla Lista proporciona un contenedor básico para guardar una lista ordenada de objetos. Lista guarda los elementos por valor, lo que significa que sirve tanto para tipos predefinidos como para instancias de clases. Por ejemplo, Lista<int> declara una lista de ints. Pero la mayoría de los patrones usan Lista para guardar punteros a objetos, como en Lista<Glifo*>. De ese modo se puede usar Lista para listas heterogéneas.

Por conveniencia, Lista también proporciona sinónimos para operaciones de pilas, lo que hace más explícito al código que usa Lista para pilas, sin necesidad de definir otra clase.

```
template <class Elemento>
class Lista {
public:
     Lista(long            tamaño            =
CAPACIDAD_LISTA_PREDETERMINADA);
     Lista(Lista&);
     ~Lista();
     Lista& operator=(const Lista&);

     long Contar() const;
     Elemento& Obtener(long indice) const;
     Elemento& Primero() const;
     Elemento& Ultimo() const;
     bool Incluye(const Elemento&) const;

     void Insertar(const Elemento&);
     void           InsertarAlPrincipio(const
Elemento&);

     void Eliminar(const Elemento&);
     void EliminarUltimo();
     void EliminarPrimero();
     void EliminarTodos();

     Elementos Cima() const;
     void Meter(const Elemento&);
     Elementos Sacar();
};
```

Las secciones siguientes describen estas operaciones en más detalle.

CONSTRUCCIÓN, DESTRUCCIÓN, INICIALIZACIÓN Y ASIGNACIÓN

Lista(long tamaño)
: inicializa la lista. El parámetro tamaño es una indicación del número inicial de elementos.

Lista(Lista&)

redefine el constructor de copia predeterminado para que los datos miembros se inicialicen adecuadamente.

~Lista()

libera las estructuras de datos internas, pero *no* los elementos de la lista. La clase no está diseñada para ser heredada; por tanto, el destructor no es virtual.

Lista& operator=(const Lista&)

implementa la operación de asignación para asignar correctamente los datos miembros.

ACCESO

Estas operaciones proporcionan el acceso básico a los elementos de la lista.

long Contar() const

devuelve el número de objetos de la lista.

Elemento& Obtener(long indice) const

devuelve el objeto del índice actual.

Elemento& Primero() const

devuelve el primer objeto de la lista.

Elemento& Ultimo() const

devuelve el último objeto de la lista.

ADICIÓN

void Insertar(const Elementos)

añade el argumento a la lista, pasando a ser el último elemento.

void InsertarAlPrincipio(const Elementos)

añade el argumento a la lista, pasando a ser el primer elemento.

BORRADO

void Eliminar(const Elementos)
elimina el elemento dado de la lista. Esta operación requiere que el tipo de los elementos de la lista admita el operador = = para la comparación.

void EliminarPrimero()
elimina el primer elemento de la lista.

void EliminarUltimo()
elimina el último elemento de la lista.

void EliminarTodos()
elimina todos los elementos de la lista.

INTERFAZ DE PILA

Elemento& Cima() const
devuelve el elemento de la cima (cuando la Lista es vista como una pila).

void Meter(const Elemento&)
mete el elemento en la pila.

Elemento& Sacar()
extrae el elemento de la cima de la pila.

C.2. ITERADOR

Iterador es una clase abstracta que define una interfaz de recorrido para agregados.

```
template <class Elemento> class Iterador {
```

```
public:
   virtual void Primero() = 0;
   virtual void Siguiente() = 0;
   virtual bool HaTerminado() const = 0;
   virtual Elemento ElementoActual() const =
0;
protected:
   Iterador();
};
```

Las operaciones hacen lo siguiente:

virtual void Primero()
posiciona el iterador sobre el primer objeto del agregado.

virtual void Siguiente()
posiciona el iterador en el siguiente objeto de la secuencia.

virtual bool HaTerminado() const
devuelve true cuando no hay más objetos en la secuencia.

virtual Elemento ElementoActual() const
devuelve el objeto situado en la posición actual de la secuencia.

C.3. ITERADORLISTA

IteradorLista implementa la interfaz Iterador para recorrer objetos Lista. Su constructor recibe la lista a recorrer como argumento.

```
template <class Elemento>
class IteradorLista : public
Iterador<Elemento> {
public:
   IteradorLista(const Lista<Elemento>*
unaLista);

   virtual void Primero();
   virtual void Siguiente();
```

```
    virtual bool HaTerminado() const;
    virtual Elemento ElementoActual() const;
};
```

C.4. PUNTO

Punto representa un punto en un espacio bidimensional de coordenadas cartesianas. Punto permite cierta aritmética de vectores mínima. Las coordenadas de un Punto se definen como

```
typedef float Coord;
```

Las operaciones de Punto se explican por sí mismas.

```
class Punto {
public:
    static const Punto Cero;

    Punto(Coord x = 0.0, Coord y = 0.0);

    Coord X() const; void X(Coord x);
    Coord Y() const; void Y(Coord y);

    friend Punto operator+(const Punto&, const
Punto&);
    friend Punto operator-(const Punto&, const
Punto&);
    friend Punto operator*(const Punto&, const
Punto&);
    friend Punto operator/(const Punto&, const
Punto&);

    Punto& operator+=(const Punto&);
    Punto& operator-=(const Punto&);
    Punto& operator*=(const Punto&);
    Punto& operator/=(const Punto&);

    Punto operator-();

    friend bool operator==(const Punto&, const
Punto&);
```

```
   friend bool operator!=(const Punto&, const
Punto&);

   friend ostream& operator<<(ostream&, const
Punto&);
   friend    istream&    operator>>(istream&,
Punto&);
};
```

El miembro estático Cero representa el Punto(0, 0).

C.5. RECT

Rect representa un rectángulo alineado con el eje. Un Rect se define por un punto de origen y una dimensión (esto es, su ancho y su alto). Las operaciones de Rect son muy fáciles de entender.

```
class Rect {
public:
   static const Rect Cero;

   Rect(Coord x, Coord y, Coord ancho, Coord
alto);
   Rect(const  Punto&  origen,  const  Punto&
dimension);

   Coord Ancho() const;void Ancho(Coord);
   Coord Alto() const;void Alto(Coord);
   Coord        Izquierda()        const;void
Izquierda(Coord);
   Coord        Inferior()         const;void
Inferior(Coord);

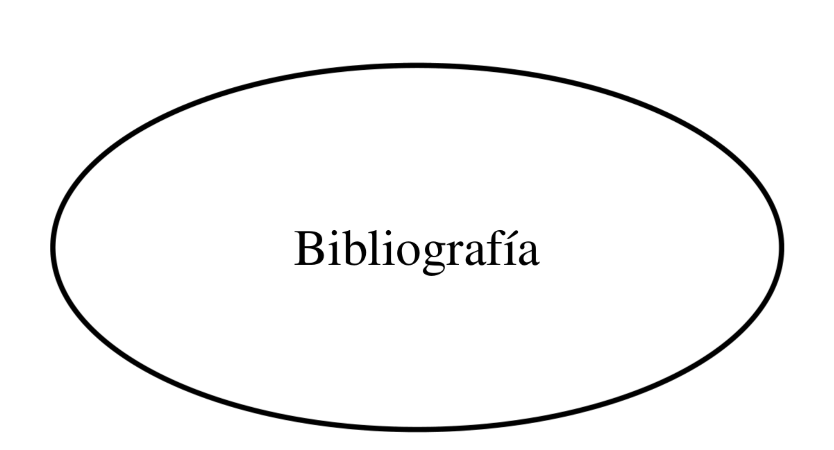
   Punto&  Origen()  const;void  Origen(const
Punto&);
   Punto&        Dimension()       const;void
Dimension(const Punto&);
   void MoverA(const Punto&);
   void Mover(const Punto&);

   bool EstaVacio() const;
   bool Contiene(const Punto&) const;
```

```
};
```

El miembro estático Cero es equivalente al rectángulo

```
Rect(Punto(0, 0), Punto(0, 0));
```


Bibliografía

[Add94] Addison-Wesley, Reading, MA. NEXTSTEP *General Reference: Release 3, Volumes 1 and 2*, 1994.

[AG90] D.B. Anderson y S. Gossain. Hierarchy evolution and the software lifecycle. In *TOOLS '90 Conference Proceedings*, pp. 41-50, París, junio de 1990. Prentice Hall.

[AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid FiksdahlKing y Shlomo Angel. *A Pattern Language*. Oxford University Press, Nueva York, 1977.

[App89] Apple Computer, Inc., Cupertino, CA. *Macintosh Programmers Workshop Pascal 3.0 Reference*, 1989.

[App92] Apple Computer, Inc., Cupertino, CA. Dylan. *An object-oriented dynamic language*, 1992.

[Arv91] *James Arvo. Graphics Gems II*. Academic Press, Boston, MA, 1991.

[AS85] B. Adelson y E. Soloway. The role of domain experience in software design. *IEEE Transactions on Software Engineering*, 11 (11):
1351-1360,
1985.

[BE93] Andreas Birrer y Thomas Eggenschwiler. Frameworks in the financial engineering domain: An experience report. In *European Conference on Object-Oriented Programming*, pp. 21-35, Kaiserslautern. Alemania, julio de 1993. Springer-Verlag.

[BJ94] Kent Beck y Ralph Johnson. Patterns generate

architectures. In *European Conference on Object-Oriented Programming*, pp. 139-149. Bologna, Italia, julio de 1994. Springer-Verlag.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, 1994. Segunda Edición.

[Bor81] A. Borning. The programming language aspects of ThingLab —a constraint-oriented simulation laboratory. ACM *Transactions on Programming Languages and Systems*, 3(4):343-387, octubre de 1981.

[Bor94] Borland International, Inc., Scotts Valley, CA. *A Technical Comparison of Borland ObjectWindows 2.0 and Microsoft MFC 2.5*, 1994.

[BV90] Grady Booch and Michael Vilot. The design of the C ++ Booch components. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pp. 1-11, Ottawa, Canada, octubre de 1990. ACM Press.

[Cal93] Paul R. Calder. *Building User Interfaces with Lightweight Objects*. Tesis Doctoral, Universidad de Stanford, 1993.

[Car89] J. Carolan. *Constructing bullet-proof classes. In Proceedings C++ at Work '89*. SIGS Publications, 1989.

[Car92] Tom Cargill. *C++ Programming Style*. Addison-Wesley, Reading, MA, 1992.

[CIRM93] Roy H. Campbell. Nayeem Islam, David Raila y Peter Madeany. Designing and implementing Choices: An object-oriented system in C++. *Communications of the ACM*, 36(9):
117-126,
septiembre de 1993.

[CL90] Paul R. Calder y Mark A. Linton. Glyphs: Flyweight objects for user interfaces. In *ACM User Interface Software Technologies Conference*, pp. 92-101, Snowbird, UT, octubre de 1990.

[CL92] Paul R. Calder y Mark A. Linton. The object-oriented implementation of a document editor. En *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pp. 154-165, Vancouver, British Columbia, Canadá, octubre de 1992. ACM Press.

[Coa92] Peter Coad. Object-oriented patterns. *Communications of the ACM*. 35(9): 152-159, septiembre de 1992.

[Coo92] William R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pp. 1-15, Vancouver, British Columbia, Canadá, octubre de 1992. ACM Press.

[Cop92] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, MA. 1992.

[Cur89] Bill Curtis. Cognitive issues in reusing software artifacts. En Ted J. Biggerstaff y Alan J. Perlis, editores, *Software Reusability, Volume II: Applications and Experience*, pp. 269-287. Addison-Wesley, Reading, MA, 1989.

[dCLF93] Dennis de Champeaux, Doug Lea y Penelope Faure. *Object-Oriented System Development*. Addison-Wesley, Reading, MA, 1993.

[Deu89] L. Peter Deutsch. Design reuse and frameworks in the Smalltalk-80 system. En Ted J. Biggerstaff y Alan J. Perlis, editores. *Software Reusability, Volume II: Applications and Experience*, pp. 57-71. Addison-Wesley,

Reading, MA, 1989.

[Ede92] D. R. Edelson, Smart pointers: They're smart, but they're not pointers. En *Proceedings of the 1992 USENIX C++ Conference*, pp. 1-19, Portland, OR, agosto de 1992. USENIX Association.

[EG92] Thomas Eggenschwiler y Erich Gamma. The ET++SwapsManager: Using object technology in the financial engineering domain. En *Object-Oriented Programming Systems. Languages, and Applications Conference Proceedings*, pp. 166-178, Vancouver, British Columbia, Canadá, octubre de 1992. ACM Press.

[ES90] Margaret A. Ellis y Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[Foo92] Brian Foote. A fractal model of the lifecycles of reusable objects. *OOPSLA '92 Workshop on Reuse*, octubre de 1992. Vancouver, British Columbia, Canada.

[GA89] S. Gossain y D.B. Anderson. Designing a class hierarchy for domain representation and reusability. In *TOOLS '89 Conference Proceedings*, pp. 201-210, CNIT Pans —La Defense, Francia, noviembre de 1989. Prentice Hall.

[Gam91] Erich Gamma. *Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools* (en alemán). Tesis Doctoral, Universidad de Zurich Instituí für Informatik, 1991.

[Gam92] Erich Gamma. *Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools* (en alemán). Springer-Verlag, Berlin, 1992.

[Gla90] Andrew Glassner. *Graphics Gems*. Academic Press, Boston, MA, 1990.

[GM92] M. Graham y E. Mettala. The Domain-Specific Software Architecture Program. En *Proceedings of DARPA Software Technology Conference*, 1992, pp. 204-210, abril de 1992. Publicado también en CrossTalk, The Journal of Defense Software Engineering, pp. 19-21, 32, octubre de 1992.

[GR83] Adele J. Goldberg y David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[HHMV92] Richard Helm, Tien Huynh, Kim Marriott y John Vlissides. An object-oriented architecture for constraint-based graphical editing. En *Proceedings of the Third Eurographics Workshop on Object-Oriented Graphics*, pp. 1-22, Champéry, Suiza, octubre de 1992. También se encuentra disponible como el informe técnico RC 18524 (79392) de la División de Investigación de IBM.

[H087] Daniel C. Halbert y Patrick D.
O'Brien.
Object-oriented development. *IEEE Software*, 4(5):71-79, septiembre de 1987.

[ION94] IONA Technologies, Ltd., Dublin, Irlanda. *Guide for Orbix. Version 1.2*
Programmer's
, 1994.

[JCJ092] Ivar Jacobson, Magnus Christerson, Patrik Jonsson y Gunnar Overgaard. *Object-Oriented Software Engineering —A Use Case Driven Approach*. Addison-Wesley, Wokingham, Inglaterra, 1992.

[JF88] Ralph E. Johnson y Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*,

1(2):22-35, junio/julio de 1988.

[JML92] Ralph E. Johnson, Carl McConnell y J. Michael Lake. The RTL system: A framework for code optimization. En Robert Giegerich y Susan L. Graham, editores, *Code Generation-Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation,* pp. 255-274, Dagstuhl, Alemania, 1992. Springer-Verlag.

[Joh92] Ralph Johnson. Documenting frameworks using patterns. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pp. 63-76, Vancouver, British Columbia, Canadá, octubre de 1992. ACM Press.

[JZ9I] Ralph E. Johnson y Jonathan Zweig. Delegation in C ++. *Journal of Object-Oriented Programming,* 4(11):22-35, noviembre de 1991.

[Kir92] David Kirk. *Graphics Gems III*. Harcourt, Brace, Jovanovich. Boston. MA, 1992.

[Knu73] Donald E. Knuth. *The Art of Computer Programming,* Volumes 1, 2, and 3. Addison-Wesley, Reading, MA, 1973.

[Knu84] Donald E. Knuth. *The TeX book*. Addison-Wesley, Reading, MA, 1984.

[Kof93] Thomas Kofler. Robust iterators in ET++. *Structured Programming,* 14:62-85, marzo de 1993.

[KP88] Glenn E. Krasner y Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming,* 1 (3):26-49, agosto/septiembre 1988.

[LaL94] Wilf LaLonde. *Discovering Smalltalk*. Benjamin/ Cummings, Redwood City, CA, 1994.

[LCI+92] Mark Linton, Paul Calder, John Interrante, Steven

Tang y John Vlissides. *Interviews Reference Manual*. CSL, Universidad de Stanford, edición 3.1, 1992.

[Lea88] Doug Lea, libg++, the GNU C++ library. En *Proceedings of the 1988 USENIX C++ Conference*, pp. 243-256, Denver, CO, octubre de 1988. USENIX Association.

[LG86] Barbara Liskov y John Guttag. *Abstraction and Specification in Program Development*. McGraw-Hill, Nueva York, 1986.

[Lie85] Henry Lieberman. There's more to menu systems than meets the screen. En *SIGGRAPH Computer Graphics*, pp. 181-189, San Francisco, CA, julio de 1985.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. En *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pp. 214-223, Portland, OR, noviembre de 1986.

[Lin92] Mark A. Linton. Encapsulating a C++ library. En *Proceedings of the 1992 USENIX C++ Conference*, pp. 57-66, Portland, OR, agosto de 1992. ACM Press.

[LP93] Mark Linton y Chuck Price. Building distributed user interfaces with Fresco. En *Proceedings of the 7th X Technical Conference*, pp. 77-87, Boston, MA. enero de 1993.

[LR93] Daniel C. Lynch y Marshall T. Rose. *Internet System Handbook*. Addison-Wesley, Reading, MA, 1993.

[LVC89] Mark A. Linton, John M. Vlissides y Paul R. Calder. Composing user interfaces with Interviews. *Computer*, 22(2):8-22, febrero de 1989.

[Mar91] Bruce Martin. The separation of interface and implementation in C++. En *Proceedings of the 1991 USENIX C++ Conference*, pp. 51-63, Washington, D.C., abril de 1991. USENIX Association.

[McC87] Paul McCullough. Transparent forwarding: First steps. En *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pp. 331-341, Orlando, FL, octubre de 1987. ACM Press.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ, 1988.

[Mur93] Robert B. Murray. *C++ Strategies and Tactics*. Addison-Wesley, Reading, MA, 1993.

[OJ90] William F. Opdyke y Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. En *SOOPPA Conference Proceedings*, pp. 145-161, Marist College, Poughkeepsie, NY, septiembre de 1990. ACM Press.

[OJ93] William F. Opdyke y Ralph E. Johnson. Creating abstract superclasses by refactoring. En *Proceedings of the 21st Annual Computer Science Conference (ACM CSC '93)*, pp. 66-73, Indianapolis, IN, febrero de 1993.

[P+88] Andrew J. Palay et al. The Andrew Toolkit: An overview. En *Proceedings of the 1988 Winter USENIX Technical Conference*, pp. 9-21, Dallas, TX, febrero de 1988. USENIX Association.

[Par90] ParcPlace Systems, Mountain View, CA. *ObjectWorks\Smalltalk Release 4 Users Guide*, 1990.

[Pas86] Geoffrey A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. En *Object-Oriented Programming Systems, Languages, and Applications*

Conference Proceedings, pp. 341-346, Portland, OR, octubre de 1986. ACM Press.

[Pug90] William Pugh. Skiplists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668-676, junio de 1990.

[RBP + 91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy y William Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Rum94] James Rumbaugh. The life of an object model: How the object model changes during development. *Journal of Object-Oriented Programming*, 7(1):24-32, marzo/abril de 1994.

[SE84] Elliot Soloway y Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5):595-609, septiembre de 1984.

[Sha90] Yen-Ping Shan. MoDE: A UIMS for Smalltalk. En *ACM OOPSLAJECOOP '90 Conference Proceedings*, pp. 258-268, Ottawa, Ontario, Canadá, octubre de 1990. ACM Press.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented languages. En *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pp. 38-45, Portland, OR, noviembre de 1986. ACM Press.

[SS86] James C. Spohrer y Elliot Soloway. Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, 29(7):624-632, julio de 1986.

[SS94] Douglas C. Schmidt y Tatsuya Suda. The Service Configurator Framework: An extensible architecture for dynamically configuring concurrent, multi-service network daemons. En *Proceeding of the Second International*

Workshop on Configurable Distributed Systems, pp. 190-201, Pittsburgh, PA, marzo de 1994. IEEE Computer Society.

[Str91] Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, Reading, MA, 1991. Segunda edición.

[Str93] Paul S. Strauss. IRIS Inventor, a 3D graphics toolkit. En *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pp. 192-200, Washington, D.C., septiembre de 1993. ACM Press.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA, 1994.

[Sut63] I. E. Sutherland. *Sketchpad: A Man-Machine Graphical Communication System*. Tesis Doctoral. MIT, 1963.

[Swe85] Richard E. Sweet. The Mesa programming environment. *SIGPLAN Notices*, 20(7):216-229, julio de 1985.

[Sym93a] Symantec Corporation, Cupertino, CA. *Bedrock Architecture Kit*
Developer's
, 1993.

[Sym93b] Symantec Corporation, Cupertino, CA. *THINK Class Library Guide*, 1993.

[Sza92] Duane Szafron. SPECTalk: An object-oriented data specification language. En *Technology of Object-Oriented Languages and Systems (TOOLS 8)*, pp. 123-138, Santa Bárbara, CA, agosto de 1992. Prentice Hall.

[US87] David Ungar y Randall B. Smith. Self: The power of simplicity. En *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pp.

227-242, Orlando, FL, octubre de 1987. ACM Press.

[VL88] John M. Vlissides y Mark A. Linton. Applying object-oriented design to structured graphics. En *Proceedings of the 1988 USENIX C++ Conference*, pp. 81-94, Denver, CO, octubre de 1988. USENIX Association.

[VL90] John M. Vlissides y Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237-268, julio de 1990.

[WBJ90] Rebecca Wirfs-Brock y Ralph E. Johnson. A survey of current research in object-oriented design. *Communications of the ACM*, 33(9): 104-124, 1990.

[WBWW90] Rebecca Wirfs-Brock, Brian Wilkerson y Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[WGM88] André Weinand, Erich Gamma y Rudolf Marty. ET+h - An object-oriented application framework in C++. En *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, pp. 46-57, San Diego, CA, septiembre de 1988. ACM Press.



ERICH GAMMA (Marzo 13, 1961, Zürich, Suiza). Informático suizo. Tiene un doctorado en ciencias de la computación en la Universidad de Zurich.

En 1995, tras la publicación del libro Design Patterns, Gamma ganó fama en el entorno de programación orientado a objetos y realizó una importante contribución a la difusión y el desarrollo del concepto de patrón de diseño. También es el diseñador principal de JUnit, un marco para la prueba de módulos de software (unidad de pruebas) que Java desarrolló en 1998 en colaboración con Kent Beck y Eclipse, una plataforma de desarrollo orientada a objetos cuyo desarrollo sigue siendo el principal responsable.

Como ingeniero de software, Gamma trabajó desde 1993 hasta 1995 en Taligent. Anteriormente, Gamma trabajó en el laboratorio de investigación UBILAB de UBS.

Erich Gamma es uno de los arquitectos de ET + +, una biblioteca de clases portátil de C + + diseñada para el desarrollo de aplicaciones gráficas.

Es actualmente empleado de Microsoft tras pasar por IBM Rational software y ser director del Object Technology International Zurich Lab en Zürich liderando el desarrollo de la plataforma Eclipse.



RICHARD HELM. Es un científico informático suizo conocido como coautor del libro Design Patterns: Elements para la reutilización de software de objetos. Tiene un doctorado en informática de la Universidad de Melbourne. En 1995, luego de la publicación del libro Design Patterns, Helm fue ampliamente reconocido en el entorno de programación orientado a objetos y realizó una importante contribución a la difusión y desarrollo del concepto de patrón de diseño.

Richard Helm regresó a IBM para establecer una nueva subsidiaria de Object Technology International en Australia, de la que ahora es miembro.

Anteriormente, Helm trabajó como consultor y diseñador de software en el Grupo DMR, una consultora internacional en el área de tecnología de la información.

Asimismo, trabajó en el departamento de tecnología de software del TJ Watson Research Center, IBM, donde laboró en diseño, reutilización y visualización orientados a objetos.

Autor de numerosas publicaciones científicas, Helm colabora regularmente con la revista

Dr. Dobb’s

Journal y en el pasado ha sido miembro del comité de programas de OOPSLA, la serie de conferencias más autorizada en el área de programación orientada a objetos.



RALPH JOHNSON (Noviembre 7, 1955. Estados Unidos). Es Profesor Asociado de Investigación en el Departamento de Ciencias de la Computación de la Universidad de Illinois en Urbana-Champaign. Es reconocido mundialmente por sus múltiples contribuciones de alto impacto en la Ingeniería de Software, particularmente en la Orientación a Objetos. De su grupo de investigación emergieron los primeros trabajos sobre frameworks, patrones y refactoring. También sus alumnos crearon la primera herramienta de refactoring, el Smalltalk Refactoring Browser, sobre cuya arquitectura se basan todas las herramientas de refactoring de la actualidad.



JOHN VLISSIDES (Agosto 2, 1961 Washington D. C. - Noviembre 24, 2005 Lake Mohegan, Nueva York, Estados Unidos). Fue un científico de software conocido principalmente como uno de los cuatro autores del libro Design Patterns: Elements of Reusable Object-Oriented Software.

Estudió ingeniería eléctrica en la Universidad de Virginia y en la Universidad de Stanford, donde en 1986 comenzó a trabajar como investigador. En 1991 se convirtió en miembro del personal de investigación del Centro de Investigación Thomas J. Watson (propiedad de IBM) en Hawthorne, Nueva York .

Durante su carrera, había publicado varios libros y artículos académicos, en particular sobre tecnologías orientadas a objetos, patrones de diseño y modelado de software.

Murió a los 44 años por complicaciones debidas a un tumor cerebral. En su honor, el grupo ACM SIGPLAN estableció el Premio John Vlissides en 2008.

Vlissides se refirió a sí mismo como “#4 de la Banda de los Cuatro y no lo haría de otra manera”.

Notas

[1] También se proporciona una traducción del nombre del patrón, si bien éste se deja en lengua inglesa por compatibilidad con toda la notación existente sobre patrones. *(N. del T.)* < <

[2] Literalmente, “peso mosca”. *(N. del T)* < <

[3] El término original inglés es “mixto”. *(N. del T)* < <

[4] *Templates* en el original en inglés. *(N. del T)* < <

[5] *Acquaintance* en el original en inglés. *(N. del T.)* < <

[6] Literalmente “juego de herramientas”. *(N. del T.)* < <

[7] Literalmente, “marco”. *(N. del T.)* < <

[8] Literalmente, “Lo que ve es lo que obtiene”. Hace referencia a los editores de documentos (sean de texto, de HTML, etcétera) donde lo que se ve en pantalla coincide con la que será la apariencia real del documento final (una vez impreso, mostrado en el navegador etcétera) *(N. del T.)* < <

[9] El diseño de Lexi está basado en Doc, una aplicación de edición de texto desarrollada por Calder [CL92]. < <

[10] Hemos traducido el término original inglés, *look and feel*, como "interfaz de usuario". *(N. del T.)* < <

[11] Los autores también suelen ver el documento en términos de su estructura lógica, es decir, en términos de oraciones, párrafos, secciones, subsecciones y capítulos. Para no complicar en exceso este ejemplo, nuestra representación interna no almacenará información acerca de la estructura lógica explícitamente. No obstante, la solución de diseño que describimos funciona igualmente bien para representar dicha información. < <

[12] *Glyph*, en el original en inglés. Palabra derivada del griego que podría traducirse como “grabado”. En castellano existe la palabra “petroglifo”, para indicar grabados en piedra (DRAE). Hemos tomado la decisión de traducirla como “glifo”, pese a que esta palabra no exista en español, por similitud con el término inglés y porque aquí no hace alusión a grabados en piedra. Además, es ésta la forma en que aparece traducida normalmente la palabra en informática, arquitectura, etcétera *(N. del T.)* <<

[13] Calder fue el primero en usar el término “glifo” (glyph) en este contexto (CL90). La mayoría de los editores de documentos actuales no utilizan un objeto para cada carácter, presumiblemente por razones de eficiencia. Calder demostró en su tesis que este enfoque es viable (Cal93). Nuestros glifos son menos sofisticados que los suyos, en el sentido de que los hemos restringido a jerarquías estrictas por simplicidad. Los glifos de Calder pueden compartirse para reducir costes de almacenamiento, formando así estructuras de grafos dirigidos acíclicos. Podría aplicarse el patrón Flyweight (179) para lograr el mismo efecto, pero dejaremos esto como ejercicio para el lector. < <

[14] La interfaz aquí descrita es intencionadamente mínima para mantener la discusión simple. Una interfaz completa incluiría operaciones para manejar los atributos gráficos tales como el color, la fuente y las transformaciones de coordenadas, además de operaciones para una gestión de los hijos más sofisticada. < <

[15] Un índice entero no es probablemente la mejor manera de especificar los hijos de un glifo, dependiendo de la estructura de datos que éste usa. Si el glifo almacena sus hijos en una lista enlazada, sería más eficiente un puntero a la lista. Veremos una solución mejor al problema de indexación en la Sección 2.8, cuando estudiemos el análisis de documentos. < <

[16] El usuario tendrá más que decir sobre la estructura *lógica* del documento —las oraciones, párrafos, secciones, capítulos, etcétera —. En comparación, la estructura *física* resulta menos interesante. A la mayoría de la gente no le preocupa dónde se insertan los saltos de línea en un párrafo siempre y cuando esté correctamente formateado. Lo mismo ocurre con el formateado de columnas y páginas. Por tanto, al final los usuarios sólo especifican restricciones de alto nivel de la estructura física, dejando que sea Lexi quien haga el trabajo duro de satisfacerlas. < <

[17] El componedor debe obtener los códigos de los caracteres de los glifos Carácter para poder calcular los saltos de línea. En la Sección 2.8 veremos cómo obtener esta información polimórficamente sin añadir una operación especifica de caracteres en la Interfaz de Glifo. < <

[18] *Transparent enclosure* en el original en inglés. *(N. del T.)* < <

[19] *Scroller* en el original en inglés. *(N. del T.)* < <

[20] Hemos traducido la expresión inglesa look and feel como “interfaz de usuario”. (N. del T.) < <

[21] En inglés, *widgets. (N. del T.)* < <

[22] Es decir, volver a realizar una operación a la que se dio marcha atrás (se anularon sus efectos). < <

[23] Conceptualmente, el cliente es el usuario de Lexi, pero en realidad es otro objeto (como un despachador de eventos) que gestiona las entradas del usuario. < <

[24] *Command* en el original en inglés. *(N. del T.)* <<

[25] El término original en inglés es *undoability*. Tanto éste como el verbo correspondiente, *undo,* hacen alusión, en este contexto, a la acción de dar marcha atrás a las operaciones, esto es, a la posibilidad de revertir los efectos de una operación anterior. Si bien "deshacer" es una traducción demasiado literal (el DRAE no sanciona esta acepción) se ha optado por ella al estar comúnmente aceptada en la mayoría de las aplicaciones existentes. *(N. del T.)* < <

[26] El original en inglés es *hyphenation*. Según el Webster, es la acción y efecto de conectar (dos palabras) o dividir (una palabra al final de la linea) con un guión. En este contexto significa lo segundo. En español se suele usar el término “guionado”, que no aparece en el DRAE. Nosotros lo hemos traducido generalmente como “inserción de guiones”. *(N. Del T.)* < <

[27] Podríamos usar sobrecarga de funciones para dar a cada una de estas funciones miembro el mismo nombre, puesto que ya están diferenciadas por sus parámetros. Les hemos dudo distintos nombres para resaltar sus diferencias, sobre todo cuando son llamadas. < <

[28] EstaMalEscrita implementa el algoritmo de revisión ortográfica, que no detallaremos aquí ya que lo hemos hecho independiente del diseño de Lexi. Podemos permitir algoritmos diferentes heredando de RevisorOrtografico; otra alternativa es aplicar el patrón Strategy (289) (igual que hicimos para el formateado en la Sección 2.3) para permitir diferentes algoritmos de revisión ortográfica. < <

[29] “Visitar” es un término ligeramente más general que “analizar”. Anticipa la terminología que usamos en el patrón de diseño al que estamos llegando. < <

[30] Hemos traducido la expresión original inglesa *look and feel* como “interfaz de usuario”. *(N. del T.)* < <

[31] *Widgets*, en el original en inglés. *(N. del T.)* < <

[32] Hemos traducido el término original en inglés, *widget*, referido a los distintos elementos de la interfaz de usuario, como “útil”. (N. del T) < <

[33] Lazy inicialization en el original en inglés. (N. del T.) < <

[34] Hemos traducido el término original inglés, *widget*, como “útil”. *(N. del T.)* < <

[35] CrearManipulador es un ejemplo de Factory Method (99). < <

[36] “Imp” viene de “implementación”. A pesar de que, siguiendo la convención de nominación que hemos venido empleando a lo largo del libro, tal vez habría sido más correcto “ImpVentana” como nombre de la clase, hemos preferido mantener “Imp” como sufijo, al igual que en el original en inglés, por ser éste prácticamente un convenio de programación. (N. del T.) < <

[37] Es fácil olvidarse de borrar el iterador una vez que se ha usado. El patrón Iterator muestra, cómo protegerse contra tales errores. < <

[38] *Widgets*, en el original en inglés. *(N. del T)* <<

[39] *Pool*, en el original en inglés. *(N. del T.)* < <

[40] El tiempo de búsqueda con este esquema es proporcional a la frecuencia de cambios de fuente. El caso peor en cuanto a rendimiento tiene lugar cuando se produce un cambio de fuente para cada carácter, pero eso es poco frecuente en la práctica. < <

[41] En el Código de Ejemplo mostrado anteriormente, la información de estilo se hizo extrínseca, dejando al código de carácter como el único estado intrínseco. < <

[42] Hemos traducido *look and feel* tomo “interfaz de usuario”. *(N. del T.)* < <

[43] *Véase* el patrón Abstract Factory (79) para otra aproximación a la independencia de la interfaz de usuario. < <

[44] *Widgets*, en el original en inglés. *(N. del T)* < <

[45] La implementación de objetos distribuidos en NEXSTEP [Add94] (en concreto, la clase NXProxy) usa esta técnica. La implementación redefine forward, el enganche equivalente en NEXTSTEP. < <

[46] El patrón Iterator (237) describe otro tipo de proxy en la página 243. < <

[47] Casi todas las clases tienen, en última instancia, s Object como superclase. Por tanto, ésta es lo mismo que decir “definir una clase que no tenga a Object como su superclase”. < <

[48] Widgets, en el original en inglés. (N. del T.) < <

[49] Para simplificar, omitiremos la precedencia de operadores y asumiremos que es responsabilidad de cualquier otro objeto construir el árbol sintáctico. < <

[50] Booch SE refiere a los iteradores externos e internos como iteradores activos y pasivos, respectivamente [Boo94], Los términos “activo” y “pasivo” describen el rol del cliente, no el nivel de actividad del iterador. < <

[51] Los cursores son un ejemplo sencillo del patrón Memento (261), y comparten con él muchos de sus detalles de implementación. <<

[52] Podemos hacer esta interfaz todavía más pequeña juntando Siguiente, HaTerminado y ElementoActual en una única operación que avance al siguiente objeto y lo devuelva. Sí se llega al final del recorrido esta operación devuelve un valor especial (0, por ejemplo) que marca el final de la iteración. < <

[53] Esto puede garantizarse en tiempo de compilación simplemente declarando privados los operadoras new y delata. No se necesita ninguna implementación adicional. < <

[54] La operación Recorrer de los ejemplos anteriores es un Factory Method (99) con ComprobarElemento y procesarElemento como operaciones primitivas. < <

[55] * *Widgets*, en el original en inglés. *(N. del T.)* < <

[56] *Hub*, en el original en inglés. *(N. del T.)* < <

[57] En este contexto, “recuerdo”. (N. del T) < <

[58] Nótese que nuestro ejemplo borra el objeto estado al final de la iteración. Pero no se llamará a delete en caso de que ProcesarElemento lance una excepción, lo que produciría basura. Esto es un problema en C++, pero no en Dylan, que tiene recolección de basura. Discutimos una solución a este problema anteriormente. < <

[59] Widget, en el original en inglés. (*N. del T*) < <

[60] Este ejemplo está basado en el protocolo de conexión TCP descrito por Lynch y Rose [LR93]. < <

[61] Esto hace de cada subclase de EstadoTCP un Singleton (véase el patrón Singleton (119)). < <

[62] Podríamos usar la sobrecarga de funciones para dar a estas operaciones el mismo nombre, como por ejemplo Visitar, ya que las operaciones se diferencian por el parámetro que se les pasa. Esta sobrecarga tiene sus pros y sus contras. Por un lado, refuerza el hecho de que cada operación implica el mismo análisis, sólo que sobre un argumento distinto. Por otro lado, podría hacer menos obvio qué está ocurriendo en el punto de la llamada para alguien que esté leyendo el código. Realmente se reduce a si creemos que la sobrecarga de funciones es buena o no. < <

[63] Sí podemos tener *doble*-despacho, ¿por qué no tener *triple, cuádruple* o cualquier otro número? En realidad, el doble-despacho no es más que un caso especial de **despacho múltiple**, en el cual la operación se elige en función de un número cualquiera de tipos (CLOS permite el despacho múltiple). Los lenguajes que permiten el despacho doble o múltiple ven reducida la necesidad del patrón Visitor. < <

[64] Este tema también afecta a otros tipos de patrones. Tanto el patrón Abstract Factory (79) como el Builder (89) y el Prototype (109), encapsulan el conocimiento sobre cómo se crean los objetos. El patrón Decorator (161) encapsula la responsabilidad que puede añadirse a un objeto. El patrón Bridge (141) separa una abstracción de su implementación, permitiéndolas variar de forma independiente. < <

[65] Véase “La poesía del lenguaje” [AIS+77]. < <

[66] OMT usa el término “diagrama de objetos” para referirse a los diagramas de clases. Nosotros usamos “diagrama de objetos” para referimos exclusivamente a diagramas de estructuras de objetos. < <

[67] OMT también define asociaciones entre clases, que pueden aparecer como líneas rectas entre rectángulos de clases Las asociaciones son bidireccionales. Aunque las asociaciones son adecuadas durante el análisis, creemos que son de demasiado alto nivel para expresar las relaciones de los patrones de diseño, debido simplemente a que las asociaciones deben convertirse en referencias a objetos o en punteros durante el diseño. Las referencias a objetos son intrínsecamente dirigidas y se adecúan por tanto mejor a las relaciones que nos ocupan. Por ejemplo, el Dibujo conoce a las Formas, pero éstas no saben nada del Dibujo en el que se encuentran. Esta relación no puede expresarse sólo con asociaciones. < <